5 PROTOTYPING EMBEDDED DEVICES

YOU HAVE AN idea for a *thing*, and you know that it has some sort of interactive or electronic side to it. What is the first step in turning that from a vision in your head into something in the real world?

You likely can try out a number of different parts of the behaviour in isolation, and that's a good starting point for your initial prototype. After you do some research on the Internet to find similar projects or look through the catalogues of component retailers, such as RS (`www.rs-components.com/`) or Farnell (`www.farnell.com/`), you'll have a list of possible components and modules which might let you achieve your goal.

The more you dabble in electronics and microcontrollers, the bigger your collection of spare parts and leftovers from previous projects will grow. When you sit down to try out some of your ideas, either you'll have rooted through your collection for parts which are close enough to those you identified in your research, or you'll have an assortment of freshly purchased components. Usually, it's a combination of the two.

That's the typical decider when first trying out an idea: you use what's easily to hand, partly because it's generally something you're familiar with already but also because it helps keep the costs down. Even if you know that the board you're using won't be the ideal fit for a final version, if it lets you try out some of the functionality more quickly or more cheaply, that can mean it's the right choice for now.

One of the main areas where something vastly overpowered, and in theory much more expensive, can be the right choice for prototyping is using a mobile phone, laptop, or desktop computer to develop the initial software. If you already have a phone or computer which you can use, using it for your prototype isn't actually any more expensive.

However, if you haven't been playing around with electronics already and don't have a collection of development boards gathering dust in your desk drawer, how do you choose which one to buy? In this chapter, we explain some of the differences and features of a number of popular options. Over time the list will change, but you should still be able to work out how the same criteria we discussed in the preceding chapter apply to whichever boards you are considering.

This chapter starts with a look at electronics because whatever platform you end up choosing, the rest of the circuitry that you will build to connect it to will be pretty much the same. Then we choose four different platforms that you could use as a basis for your Internet of Things prototype. They aren't the only options, but they cover the breadth of options available. By the end of the chapter you will have a good feel for the trade-offs between the different options and enough knowledge of the example boards to make a choice on which to explore further.

ELECTRONICS

Before we get stuck into the ins and outs of microcontroller and embedded computer boards, let's address some of the electronics components that you might want to connect to them.

Don't worry if you're scared of things such as having to learn soldering. You are unlikely to need it for your initial experiments. Most of the prototyping can be done on what are called *solderless breadboards*. They enable you to build components together into a circuit with just a push-fit connection, which also means you can experiment with different options quickly and easily.

When it comes to thinking about the electronics, it's useful to split them into two main categories:

- **Sensors:** Sensors are the ways of getting information *into* your device, finding out things about your surroundings.
- **Actuators:** Actuators are the *outputs* for the device—the motors, lights, and so on, which let your device do something to the outside world.

Within both categories, the electronic components can talk to the computer in a number of ways.

The simplest is through digital I/O, which has only two states: a button can either be pressed or not; or an LED can be on or off. These states are usually connected via general-purpose input/output (GPIO) pins and map a digital 0 in the processor to 0 volts in the circuit and the digital 1 to a set voltage, usually the voltage that the processor is using to run (commonly 5V or 3.3V).

If you want a more nuanced connection than just on/off, you need an analogue signal. For example, if you wire up a potentiometer to let you read in the position of a rotary knob, you will get a varying voltage, depending on the knob's location. Similarly, if you want to run a motor at a speed other than off or full-speed, you need to feed it with a voltage somewhere between 0V and its maximum rating.

Because computers are purely digital devices, you need a way to translate between the analogue voltages in the real world and the digital of the computer.

An analogue-to-digital converter (ADC) lets you measure varying voltages. Microcontrollers often have a number of these converters built in. They will convert the voltage level between 0V and a predefined maximum (often the same 5V or 3.3V the processor is running at, but sometimes a fixed value such as 1V) into a number, depending on the accuracy of the ADC. The Arduino has 10-bit ADCs, which by default measure voltages between 0 and 5V. A voltage of 0 will give a reading of 0; a voltage of 5V would read 1023 (the maximum value that can be stored in a 10-bits); and voltages in between result in readings relative to the voltage. 1V would map to 205; a reading of 512 would mean the voltage was 2.5V; and so on.

The flipside of an ADC is a DAC, or digital-to-analogue converter. DACs let you generate varying voltages from a digital value but are less common as a standard feature of microcontrollers. This is due to a technique called *pulse-width modulation* (PWM), which gives an approximation to a DAC by

rapidly turning a digital signal on and off so that the average value is the level you desire. PWM requires simpler circuitry, and for certain applications, such as fading an LED, it is actually the preferred option.

For more complicated sensors and modules, there are interfaces such as Serial Peripheral Interface (SPI) bus and Inter-Integrated Circuit (I2C). These standardised mechanisms allow modules to communicate, so sensors or things such as Ethernet modules or SD cards can interface to the microcontroller.

Naturally, we can't cover all the possible sensors and actuators available, but we list some of the more common ones here to give a flavour of what is possible.

SENSORS

Pushbuttons and switches, which are probably the simplest sensors, allow some user input. Potentiometers (both rotary and linear) and rotary encoders enable you to measure movement.

Sensing the environment is another easy option. Light-dependent resistors (LDRs) allow measurement of ambient light levels, thermistors and other temperature sensors allow you to know how warm it is, and sensors to measure humidity or moisture levels are easy to build.

Microphones obviously let you monitor sounds and audio, but piezo elements (used in certain types of microphones) can also be used to respond to vibration.

Distance-sensing modules, which work by bouncing either an infrared or ultrasonic signal off objects, are readily available and as easy to interface to as a potentiometer.

ACTUATORS

One of the simplest and yet most useful actuators is light, because it is easy to create electronically and gives an obvious output. Light-emitting diodes (LEDs) typically come in red and green but also white and other colours. RGB LEDs have a more complicated setup but allow you to mix the levels of red, green, and blue to make whatever colour of light you want. More complicated visual outputs also are available, such as LCD screens to display text or even simple graphics.

Piezo elements, as well as *responding* to vibration, can be used to *create* it, so you can use a piezo buzzer to create simple sounds and music. Alternatively, you can wire up outputs to speakers to create more complicated synthesised sounds.

Of course, for many tasks, you might also want to use components that *move* things in the real world. Solenoids can by used to create a single, sharp pushing motion, which could be useful for pushing a ball off a ledge or tapping a surface to make a musical sound.

More complicated again are motors. Stepper motors can be moved in *steps*, as the name implies. Usually, a fixed number of steps perform a full rotation. DC motors simply move at a given speed when told to. Both types of motor can be one-directional or move in both directions. Alternatively, if you want a motor that will turn to a given angle, you would need a servo. Although a servo is more controllable, it tends to have a shorter range of motion, often 180 or fewer degrees (whereas steppers and DC motors turn indefinitely). For all the kinds of motors that we've mentioned, you typically want to connect the motors to gears to alter the range of motion or convert circular movement to linear, and so on.

> *If you want to dig further into the ways of interfacing your computer or microcontroller with the real world, the "Interfacing with Hardware" page on the Arduino Playground website (`http://playground.arduino.cc//Main/InterfacingWithHardware`) is a good place to start. Although Arduino-focused, most of the suggestions will translate to other platforms with minimal changes. For a more in-depth introduction to electronics, we recommend* Electronics For Dummies *(Wiley, 2009).*

SCALING UP THE ELECTRONICS

From the perspective of the electronics, the starting point for prototyping is usually a "breadboard". This lets you push-fit components and wires to make up circuits without requiring any soldering and therefore makes experimentation easy. When you're happy with how things are wired up, it's common to solder the components onto some protoboard, which may be sufficient to make the circuit more permanent and prevent wires from going astray.

Moving beyond the protoboard option tends to involve learning how to lay out a PCB. This task isn't as difficult as it sounds, for simple circuits at least,

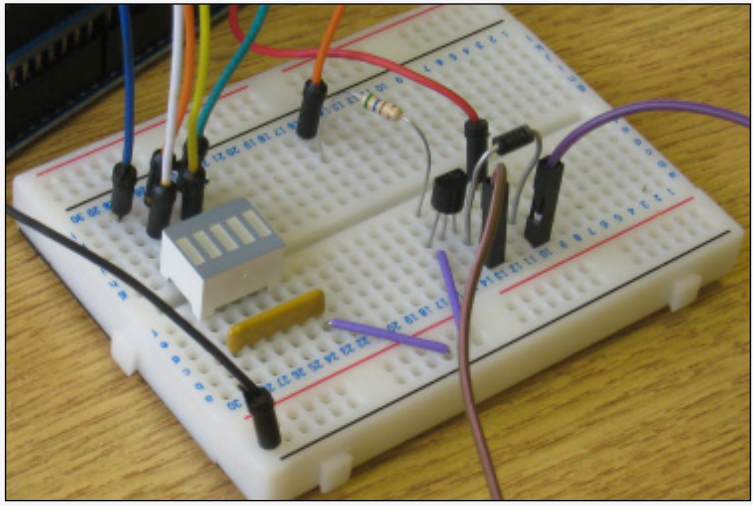
and mainly involves learning how to use a new piece of software and understanding some new terminology.

For small production runs, you'll likely use through-hole components, so called because the legs of the component go through holes in the PCB and tend to be soldered by hand. You will often create your designs as companion boards to an existing microcontroller platform—generally called *shields* in the Arduino community. This approach lets you bootstrap production without worrying about designing the entire system from scratch.

Journey to a Circuit Board

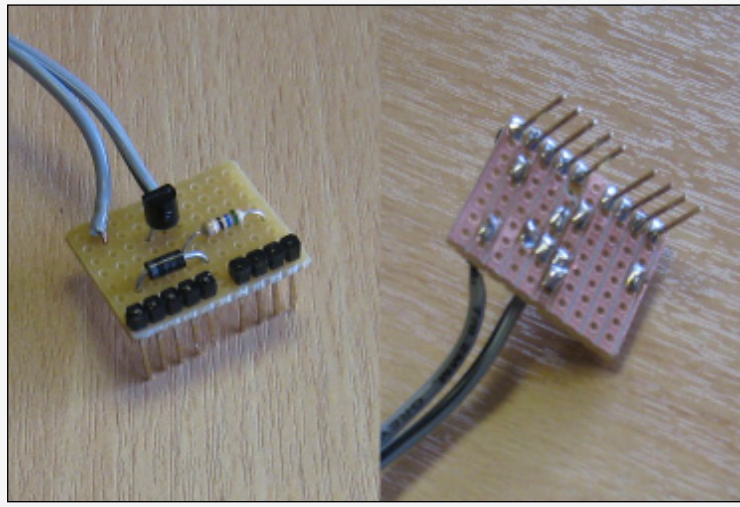
Let's look at the evolution of part of the Bubblino circuitry, from initial testing, through prototype, to finished PCB:

1. The first step in creating your circuit is generally to build it up on a breadboard. This way, you can easily reconfigure things as you decide exactly how it should be laid out.



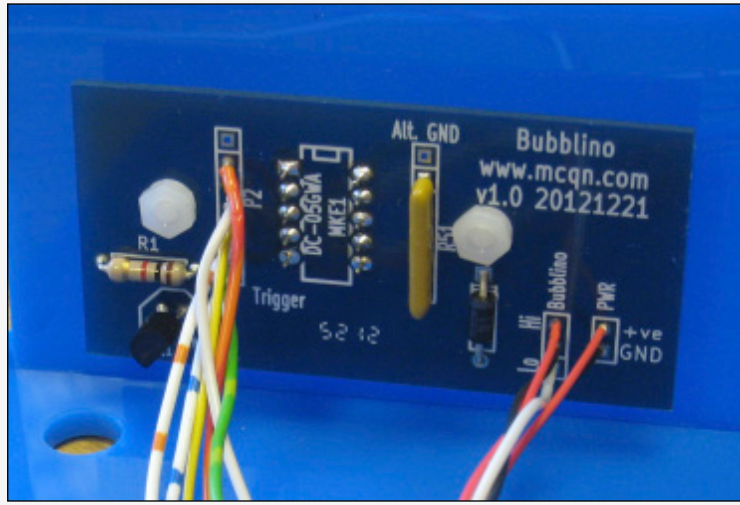
The breadboard.

2. When you are happy with how the circuit works, soldering it onto a stripboard will make the layout permanent. This means you can stop worrying about one of the wires coming loose, and if you're going to make only one copy of the circuit, that might be as far as you need take things.



The stripboard.

3. If you need to make many copies of the circuit, or if you want a professional finish, you can turn your circuit into a PCB. This makes it easier to build up the circuit because the position of each component will be labelled, there will be holes only where the components go, and there will be less chance of short circuits because the tracks between components will be protected by the solder resist.



The PCB.

When you want to scale things even further, moving to a combined board allows you to remove any unnecessary components from the microcontroller board, and switching to surface mount components—where the legs of the chips are soldered onto the same surface as the chip—eases the board's assembly with automated manufacturing lines.

PCB design and the options for manufacturing are covered in much greater detail in Chapter 10, "Moving to Manufacture".

EMBEDDED COMPUTING BASICS

The rest of this chapter examines a number of different embedded computing platforms, so it makes sense to first cover some of the concepts and terms that you will encounter along the way.

Providing background is especially important because many of you may have little or no idea about what a microcontroller is. Although we've been talking about computing power getting cheaper and more powerful, you cannot just throw a bunch of PC components into something and call it an Internet of Things product. If you've ever opened up a desktop PC, you've seen that it's a collection of discrete modules to provide different aspects of functionality. It has a main motherboard with its processor, one or two smaller circuit boards providing the RAM, and a hard disk to provide the long-term storage. So, it has a lot of components, which provide a variety of general-purpose functionality and which all take up a corresponding chunk of physical space.

MICROCONTROLLERS

Internet of Things devices take advantage of more tightly integrated and miniaturised solutions—from the most basic level of microcontrollers to more powerful system-on-chip (SoC) modules. These systems combine the processor, RAM, and storage onto a single chip, which means they are much more specialised, smaller than their PC equivalents, and also easier to build into a custom design.

These microcontrollers are the engines of countless sensors and automated factory machinery. They are the last bastions of 8-bit computing in a world that's long since moved to 32-bit and beyond. Microcontrollers are very limited in their capabilities—which is why 8-bit microcontrollers are still in use, although the price of 32-bit microcontrollers is now dropping to the level where they're starting to be edged out. Usually, they offer RAM

capabilities measured in kilobytes and storage in the tens of kilobytes. However, they can still achieve a lot despite their limitations.

You'd be forgiven if the mention of 8-bit computing and RAM measured in kilobytes gives you flashbacks to the early home computers of the 1980s such as the Commodore 64 or the Sinclair ZX Spectrum. The 8-bit microcontrollers have the same sort of internal workings and similar levels of memory to work with. There have been some improvements in the intervening years, though—the modern chips are much smaller, require less power, and run about five times faster than their 1980s counterparts.

Unlike the market for desktop computer processors, which is dominated by two manufacturers (Intel and AMD), the microcontroller market consists of many manufacturers. A better comparison is with the automotive market. In the same way that there are many different car manufacturers, each with a range of models for different uses, so there are lots of microcontroller manufacturers (Atmel, Microchip, NXP, Texas Instruments, to name a few), each with a range of chips for different applications.

The ubiquitous Arduino platform is based around Atmel's AVR ATmega family of microcontroller chips. The on-board inclusion of an assortment of GPIO pins and ADC circuitry means that microcontrollers are easy to wire up to all manner of sensors, lights, and motors. Because the devices using them are focused on performing one task, they can dispense with most of what we would term an operating system, resulting in a simpler and much slimmer code footprint than that of a SoC or PC solution.

In these systems, functions which require greater resource levels are usually provided by additional single-purpose chips which at times are more powerful than their controlling microcontroller. For example, the WizNet Ethernet chip used by the Arduino Ethernet has eight times more RAM than the Arduino itself.

SYSTEM-ON-CHIPS

In between the low-end microcontroller and a full-blown PC sits the SoC (for example, the BeagleBone or the Raspberry Pi). Like the microcontroller, these SoCs combine a processor and a number of peripherals onto a single chip but usually have more capabilities. The processors usually range from a few hundred megahertz, nudging into the gigahertz for top-end solutions, and include RAM measured in megabytes rather than kilobytes. Storage for SoC modules tends not to be included on the chip, with SD cards being a popular solution.

The greater capabilities of SoC mean that they need some sort of operating system to marshal their resources. A wide selection of embedded operating systems, both closed and open source, is available and from both specialised embedded providers and the big OS players, such as Microsoft and Linux. Again, as the price falls for increased computing power, the popularity and familiarity of options such as Linux are driving its wider adoption.

CHOOSING YOUR PLATFORM

How to choose the *right* platform for your Internet of Things device is as easy a question to answer as working out the meaning of life. This isn't to say that it's an impossible question—more that there are almost as many answers as there are possible devices. The platform you choose depends on the particular blend of price, performance, and capabilities that suit what you're trying to achieve. And just because you settle on one solution, that doesn't mean somebody else wouldn't have chosen a completely different set of options to solve the same problem.

Start by choosing a platform to prototype in. The following sections discuss some of the factors that you need to weigh—and possibly play off against each other—when deciding how to build your device.

> *We cover the decisions that you need to make when scaling up both later in this chapter and in Chapter 10.*

Processor Speed

The processor speed, or clock speed, of your processor tells you how fast it can process the individual instructions in the machine code for the program it's running. Naturally, a faster processor speed means that it can execute instructions more quickly.

The clock speed is still the simplest proxy for raw computing power, but it isn't the only one. You might also make a comparison based on millions of instructions per second (MIPS), depending on what numbers are being reported in the datasheet or specification for the platforms you are comparing.

Some processors may lack hardware support for floating-point calculations, so if the code involves a lot of complicated mathematics, a by-the-numbers slower processor with hardware floating-point support could be faster than a slightly higher performance processor without it.

Generally, you will use the processor speed as one of a number of factors when weighing up similar systems. Microcontrollers tend to be clocked at speeds in the tens of MHz, whereas SoCs run at hundreds of MHz or possibly low GHz.

If your project doesn't require heavyweight processing—for example, if it needs only networking and fairly basic sensing—then some sort of microcontroller will be fast enough. If your device will be crunching lots of data—for example, processing video in real time—then you'll be looking at a SoC platform.

RAM

RAM provides the working memory for the system. If you have more RAM, you may be able to do more things or have more flexibility over your choice of coding algorithm. If you're handling large datasets on the device, that could govern how much space you need. You can often find ways to work around memory limitations, either in code (see Chapter 8, "Techniques for Writing Embedded Code") or by handing off processing to an online service (see Chapter 7, "Prototyping Online Components").

It is difficult to give exact guidelines to the amount of RAM you will need, as it will vary from project to project. However, microcontrollers with less than 1KB of RAM are unlikely to be of interest, and if you want to run standard encryption protocols, you will need at least 4KB, and preferably more.

For SoC boards, particularly if you plan to run Linux as the operating system, we recommend at least 256MB.

Networking

How your device connects to the rest of the world is a key consideration for Internet of Things products. Wired Ethernet is often the simplest for the user—generally plug and play—and cheapest, but it requires a physical cable. Wireless solutions obviously avoid that requirement but introduce a more complicated configuration.

WiFi is the most widely deployed to provide an existing infrastructure for connections, but it can be more expensive and less optimized for power consumption than some of its competitors.

Other short-range wireless can offer better power-consumption profiles or costs than WiFi but usually with the trade-off of lower bandwidth. ZigBee is

one such technology, aimed particularly at sensor networks and scenarios such as home automation. The recent Bluetooth LE protocol (also known as Bluetooth 4.0) has a very low power-consumption profile similar to ZigBee's and could see more rapid adoption due to its inclusion into standard Bluetooth chips included in phones and laptops. There is, of course, the existing Bluetooth standard as another possible choice. And at the boring-but-very-cheap end of the market sit long-established options such as RFM12B which operate in the 434 MHz radio spectrum, rather than the 2.4 GHz range of the other options we've discussed.

For remote or outdoor deployment, little beats simply using the mobile phone networks. For low-bandwidth, higher-latency communication, you could use something as basic as SMS; for higher data rates, you will use the same data connections, like 3G, as a smartphone.

USB

If your device can rely on a more powerful computer being nearby, tethering to it via USB can be an easy way to provide both power and networking. You can buy some of the microcontrollers in versions which include support for USB, so choosing one of them reduces the need for an extra chip in your circuit.

Instead of the microcontroller presenting itself as a device, some can also act as the USB "host". This configuration lets you connect items that would normally expect to be connected to a computer—devices such as phones, for example, using the Android ADK, additional storage capacity, or WiFi dongles.

Devices such as WiFi dongles often depend on additional software on the host system, such as networking stacks, and so are better suited to the more computer-like option of SoC.

Power Consumption

Faster processors are often more power hungry than slower ones. For devices which might be portable or rely on an unconventional power supply (batteries, solar power) depending on where they are installed, power consumption may be an issue. Even with access to mains electricity, the power consumption may be something to consider because lower consumption may be a desirable feature.

However, processors may have a minimal power-consumption sleep mode. This mode may allow you to use a faster processor to quickly perform

operations and then return to low-power sleep. Therefore, a more powerful processor may *not* be a disadvantage even in a low-power embedded device.

Interfacing with Sensors and Other Circuitry

In addition to talking to the Internet, your device needs to interact with something else—either sensors to gather data about its environment; or motors, LEDs, screens, and so on, to provide output. You could connect to the circuitry through some sort of peripheral bus—SPI and I2C being common ones—or through ADC or DAC modules to read or write varying voltages; or through generic GPIO pins, which provide digital on/off inputs or outputs. Different microcontrollers or SoC solutions offer different mixtures of these interfaces in differing numbers.

Physical Size and Form Factor

The continual improvement in manufacturing techniques for silicon chips means that we've long passed the point where the limiting factor in the size of a chip is the amount of space required for all the transistors and other components that make up the circuitry on the silicon. Nowadays, the size is governed by the number of connections it needs to make to the surrounding components on the PCB.

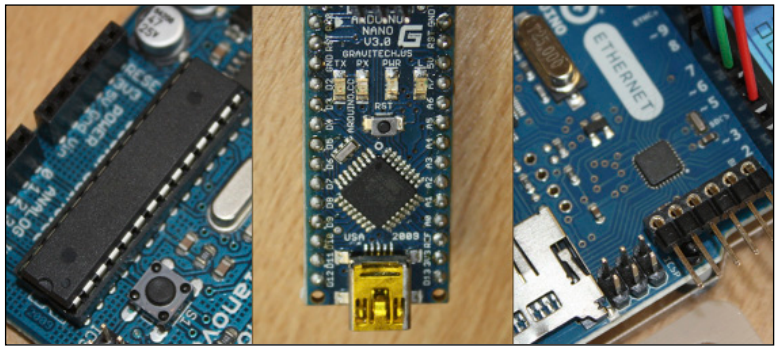
With the traditional through-hole design, most commonly used for home-made circuits, the legs of the chip are usually spaced at 0.1" intervals. Even if your chip has relatively few connections to the surrounding circuit—16 pins is nothing for such a chip—you will end up with over 1.5" (~4cm) for the perimeter of your chip. More complex chips can easily run to over a hundred connections; finding room for a chip with a 10" (25cm) perimeter might be a bit tricky!

You can pack the legs closer together with surface-mount technology because it doesn't require holes to be drilled in the board for connections. Combining that with the trick of hiding some of the connections on the underside of the chip means that it is possible to use the complex designs without resorting to PCBs the size of a table.

The limit to the size that each connection can be reduced to is then governed by the capabilities and tolerances of your manufacturing process. Some surface-mount designs are big enough for home-etched PCBs and can be hand-soldered. Others require professionally produced PCBs and accurate pick-and-place machines to locate them correctly.

Due to these trade-offs in size versus manufacturing complexity, many chip designs are available in a number of different form factors, known as *packages*. This lets the circuit designer choose the form that best suits his particular application.

All three chips pictured in the following figure provide identical functionality because they are all AVR ATmega328 microcontrollers. The one on the left is the through-hole package, mounted here in a socket so that it can be swapped out without soldering. The two others are surface mount, in two different packages, showing the reduction in size but at the expense of ease of soldering.



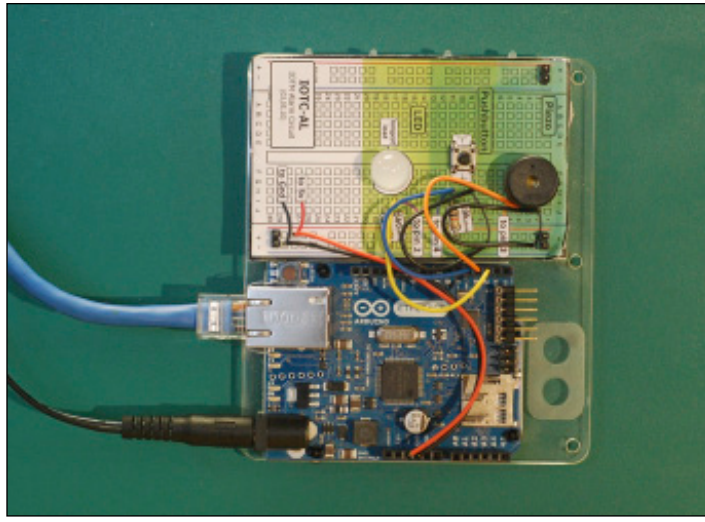
Through-hole versus surface-mount ATmega328 chips.

Looking at the ATmega328 leads us nicely into comparing some specific embedded computing platforms. We can start with a look at one which so popularised the ATmega328 that a couple of years ago it led to a worldwide shortage of the chip in the through-hole package, as for a short period demand outstripped supply.

ARDUINO

Without a doubt, the poster child for the Internet of Things, and physical computing in general, is the Arduino.

These days the Arduino project covers a number of microcontroller boards, but its birth was in Ivrea in Northern Italy in 2005. A group from the Interaction Design Institute Ivrea (IDII) wanted a board for its design students to use to build interactive projects. An assortment of boards was around at that time, but they tended to be expensive, hard to use, or both.



An Arduino Ethernet board, plugged in, wired up to a circuit and ready for use.

So, the team put together a board which was cheap to buy—around £20—and included an onboard serial connection to allow it to be easily programmed. Combined with an extension of the Wiring software environment, it made a huge impact on the world of physical computing.

Wiring: Sketching in Hardware

Another child of the IDII is the Wiring project. In the summer of 2003, Hernando Barragán started a project to make it easier to experiment with electronics and hardware. As the project website (`http://wiring.org.co/about.html`) puts it:

"The idea is to write a few lines of code, connect a few electronic components to the hardware of choice and observe how a light turns on when person approaches to it, write a few more lines add another sensor and see how this light changes when the illumination level in a room decreases.

This process is called sketching with hardware—a way to explore lots of ideas very quickly, by selecting the more interesting ones, refining them, and producing prototypes in an iterative process."

The Wiring platform provides an abstraction layer over the hardware, so the users need not worry about the exact way to, say, turn on a GPIO pin, and can focus on the problem they're trying to explore or solve.

That abstraction also enables the platform to run on a variety of hardware boards. There have been a number of Wiring boards since the project started, although they have been eclipsed by the runaway success of the project that took the Wiring platform and targeted a lower-end and cheaper AVR processor: the Arduino project.

A decision early on to make the code and schematics open source meant that the Arduino board could outlive the demise of the IDII and flourish. It also meant that people could adapt and extend the platform to suit their own needs.

As a result, an entire ecosystem of boards, add-ons, and related kits has flourished. The Arduino team's focus on simplicity rather than raw performance for the code has made the Arduino the board of choice in almost every beginner's physical computing project, and the open source ethos has encouraged the community to share circuit diagrams, parts lists, and source code. It's almost the case that whatever your project idea is, a quick search on Google for it, in combination with the word "Arduino", will throw up at least one project that can help bootstrap what you're trying to achieve. If you prefer learning from a book, we recommend picking up a copy of *Arduino For Dummies*, by John Nussey (Wiley, 2013).

The "standard" Arduino board has gone through a number of iterations: Arduino NG, Diecimila, Duemilanove, and Uno.

The Uno features an ATmega328 microcontroller and a USB socket for connection to a computer. It has 32KB of storage and 2KB of RAM, but don't let those meagre amounts of memory put you off; you can achieve a surprising amount despite the limitations.

The Uno also provides 14 GPIO pins (of which 6 can also provide PWM output) and 6 10-bit resolution ADC pins. The ATmega's serial port is made available through both the IO pins, and, via an additional chip, the USB connector.

If you need more space or a greater number of inputs or outputs, look at the Arduino Mega 2560. It marries a more powerful ATmega microcontroller to the same software environment, providing 256KB of Flash storage, 8KB of RAM, three more serial ports, a massive 54 GPIO pins (14 of those also capable of PWM) and 16 ADCs. Alternatively, the more recent Arduino Due has a 32-bit ARM core microcontroller and is the first of the Arduino boards to use this architecture. Its specs are similar to the Mega's, although it ups the RAM to 96KB.

DEVELOPING ON THE ARDUINO

More than just specs, the experience of working with a board may be the most important factor, at least at the prototyping stage. As previously mentioned, the Arduino is optimised for simplicity, and this is evident from the way it is packaged for use. Using a single USB cable, you can not only

power the board but also push your code onto it, and (if needed) communicate with it—for example, for debugging or to use the computer to store data retrieved by the sensors connected to the Arduino.

Of course, although the Arduino was at the forefront of this drive for ease-of-use, most of the microcontrollers we look at in this chapter attempt the same, some less successfully than others.

Integrated Development Environment

You usually develop against the Arduino using the integrated development environment (IDE) that the team supply at `http://arduino.cc`. Although this is a fully functional IDE, based on the one used for the Processing language (`http://processing.org/`), it is very simple to use. Most Arduino projects consist of a single file of code, so you can think of the IDE mostly as a simple file editor. The controls that you use the most are those to check the code (by compiling it) or to push code to the board.

Pushing Code

Connecting to the board should be relatively straightforward via a USB cable. Sometimes you might have issues with the drivers (especially on some versions of Windows) or with permissions on the USB port (some Linux packages for drivers don't add you to the dialout group), but they are usually swiftly resolved once and for good. After this, you need to choose the correct serial port (which you can discover from system logs or select by trial and error) and the board type (from the appropriate menus, you may need to look carefully at the labelling on your board and its CPU to determine which option to select).

When your setup is correct, the process of pushing code is generally simple: first, the code is checked and compiled, with any compilation errors reported to you. If the code compiles successfully, it gets transferred to the Arduino and stored in its flash memory. At this point, the Arduino reboots and starts running the new code.

Operating System

The Arduino doesn't, by default, run an OS as such, only the bootloader, which simplifies the code-pushing process described previously. When you switch on the board, it simply runs the code that you have compiled until the board is switched off again (or the code crashes).

It is, however, possible to upload an OS to the Arduino, usually a lightweight real-time operating system (RTOS) such as FreeRTOS/DuinOS. The main advantage of one of these operating systems is their built-in support for multitasking. However, for many purposes, you can achieve reasonable results with a simpler task-dispatching library.

If you dislike the simple life, it is even possible to compile code without using the IDE but by using the toolset for the Arduino's chip—for example, for all the boards until the recent ARM-based Due, the `avr-gcc` toolset.

The `avr-gcc` toolset (`www.nongnu.org/avr-libc/`) is the collection of programs that let you compile code to run on the AVR chips used by the rest of the Arduino boards and flash the resultant executable to the chip. It is used by the Arduino IDE behind the scenes but can be used directly, as well.

Language

The language usually used for Arduino is a slightly modified dialect of C++ derived from the Wiring platform. It includes some libraries used to read and write data from the I/O pins provided on the Arduino and to do some basic handling for "interrupts" (a way of doing multitasking, at a very low level). This variant of C++ tries to be forgiving about the ordering of code; for example, it allows you to call functions before they are defined. This alteration is just a nicety, but it is useful to be able to order things in a way that the code is easy to read and maintain, given that it tends to be written in a single file.

The code needs to provide only two routines:

- `setup()`: This routine is run once when the board first boots. You could use it to set the modes of I/O pins to input or output or to prepare a data structure which will be used throughout the program.
- `loop()`: This routine is run repeatedly in a tight loop while the Arduino is switched on. Typically, you might check some input, do some calculation on it, and perhaps do some output in response.

To avoid getting into the details of programming languages in this chapter, we just compare a simple example across all the boards—blinking a single LED:

```
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;
```

```
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off
  delay(1000);               // wait for a second
}
```

Reading through this code, you'll see that the `setup()` function does very little; it just sets up that pin number 13 is the one we're going to control (because it is wired up to an LED).

Then, in `loop()`, the LED is turned on and then off, with a delay of a second between each flick of the (electronic) switch. With the way that the Arduino environment works, whenever it reaches the end of one cycle—on; wait a second; off; wait a second—and drops out of the `loop()` function, it simply calls `loop()` again to repeat the process.

Debugging

Because C++ is a compiled language, a fair number of errors, such as bad syntax or failure to declare variables, are caught at compilation time. Because this happens on your computer, you have ample opportunity to get detailed and possibly helpful information from the compiler about what the problem is.

Although you need some debugging experience to be able to identify certain compiler errors, others, like this one, are relatively easy to understand:

```
Blink.cpp: In function 'void loop()':Blink:21:
error:'digitalWritee' was not declared in this scope
```

On line 21, in the function `loop()`, we deliberately misspelled the call to `digitalWrite`.

When the code is pushed to the Arduino, the rules of the game change, however. Because the Arduino isn't generally connected to a screen, it is hard for it to tell you when something goes wrong. Even if the code compiled

successfully, certain errors still happen. An error could be raised that can't be handled, such as a division by zero, or trying to access the tenth element of a 9-element list. Or perhaps your program leaks memory and eventually just stops working. Or (and worse) a programming error might make the code continue to work dutifully but give entirely the wrong results.

If Bubblino stops blowing bubbles, how can we distinguish between the following cases?

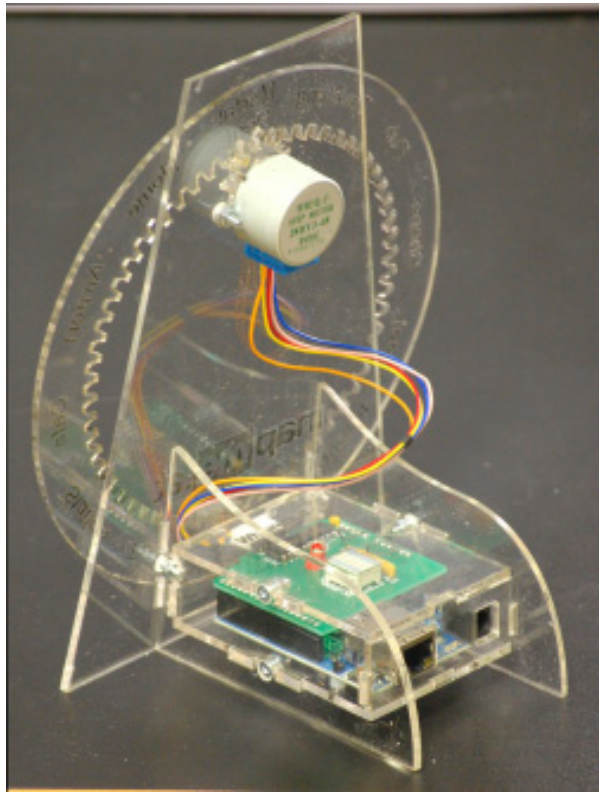
- Nobody has mentioned us on Twitter.
- The Twitter search API has stopped working.
- Bubblino can't connect to the Internet.
- Bubblino has crashed due to a programming error.
- Bubblino is working, but the motor of the bubble machine has failed.
- Bubblino is powered off.

Adrian likes to joke that he can debug many problems by looking at the flashing lights at Bubblino's Ethernet port, which flashes while Bubblino connects to DNS and again when it connects to Twitter's search API, and so on. (He also jokes that we can discount the "programming error" option and that the main reason the motor would fail is that Hakim has poured bubble mix into the wrong hole. Again.) But while this approach might help distinguish two of the preceding cases, it doesn't help with the others and isn't useful if you are releasing the product into a mass market!

The first commercially available version of the WhereDial has a bank of half a dozen LEDs specifically for consumer-level debugging. In the case of an error, the pattern of lights showing may help customers fix their problem or help flesh out details for a support request.

Runtime programming errors may be tricky to trap because although the C++ language has exception handling, the `avr-gcc` compiler doesn't support it (probably due to the relatively high memory "cost" of handling exceptions); so the Arduino platform doesn't let you use the usual `try... catch...` logic.

Effectively, this means that you need to check your data before using it: if a number might conceivably be zero, check that before trying to divide by it. Test that your indexes are within bounds. To avoid memory leaks, look at the tips on writing code for embedded devices in Chapter 8, "Techniques for Writing Embedded Code".



Rear view of a transparent WhereDial. The bank of LEDs can be seen in the middle of the green board, next to the red "error" LED.

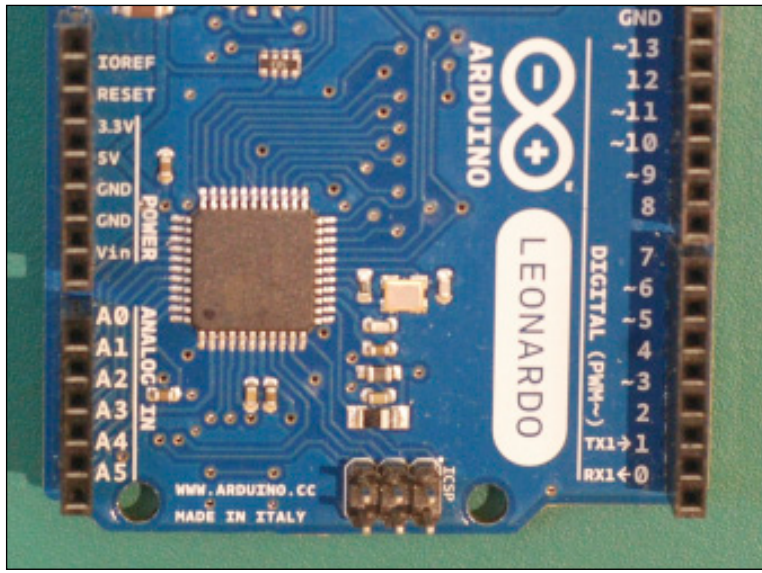
But code isn't, in general, created perfect: in the meantime you need ways to identify where the errors are occurring so that you can bullet-proof them for next time. In the absence of a screen, the Arduino allows you to write information over the USB cable using `Serial.write()`. Although you can use the facility to communicate all kinds of data, debugging information can be particularly useful. The Arduino IDE provides a serial monitor which echoes the data that the Arduino has sent over the USB cable. This could include any textual information, such as logging information, comments, and details about the data that the Arduino is receiving and processing (to double-check that your calculations are doing the right thing).

SOME NOTES ON THE HARDWARE

The Arduino exposes a number of GPIO pins and is usually supplied with "headers" (plastic strips that sit on the pin holes, that provide a convenient

solderless connection for wires, especially with a "jumper" connection). The headers are optimised for prototyping and for being able to change the purpose of the Arduino easily.

Each pin is clearly labelled on the controller board. The details of pins vary from the smaller boards such as the Nano, the classic form factor of the Uno, and the larger boards such as the Mega or the Due. In general, you have power outputs such as 5 volts or 3.3 volts (usually labelled 5V and 3V3, or perhaps just 3V), one or more electric ground connections (GND), numbered digital pins, and numbered analogue pins prefixed with an *A*.



Close-up of an Arduino Leonardo board. Note the labelling of the power and analogue input connections.

You can power the Arduino using a USB connection from your computer. This capability is usually quite convenient during prototyping because you need the serial connection in any case to program the board. The Arduino also has a socket for an external power supply, which you might be more likely to use if you distribute the project. Either way should be capable of powering the microcontroller and the usual electronics that you might attach to it. (In the case of larger items, such as motors, you may have to attach external power and make that available selectively to the component using transistors.)

Outside of the standard boards, a number of them are focused on a particular niche application—for example, the Arduino Ethernet has an on-board Ethernet chip and trades the USB socket for an Ethernet one, making it

easier to hook up to the Internet. This is obviously a strong contender for a useful board for Internet of Things projects.

The LilyPad has an entirely different specialism, as it has a flattened form (shaped, as the name suggests, like a flower with the I/O capabilities exposed on its "petals") and is designed to make it easy to wire up with conductive thread, and so a boon for wearable technology projects.

Choosing one of the specialist boards isn't the only way to extend the capabilities of your Arduino. Most of the boards share the same layout of the assorted GPIO, ADC, and power pins, and you are able to piggyback an additional circuit board on top of the Arduino which can contain all manner of componentry to give the Arduino extra capabilities.

In the Arduino world, these add-on boards are called *shields*, perhaps because they cover the actual board as if protecting it.

Some shields provide networking capabilities—Ethernet, WiFi, or Zigbee wireless, for example. Motor shields make it simple to connect motors and servos; there are shields to hook up mobile phone LCD screens; others to provide capacitive sensing; others to play MP3 files or WAV files from an SD card; and all manner of other possibilities—so much so that an entire website, `http://shieldlist.org/`, is dedicated to comparing and documenting them.

In terms of functionality, a standard Arduino with an Ethernet shield is equivalent to an Arduino Ethernet. However, the latter is thinner (because it has all the components laid out on a single board) but loses the convenient USB connection. (You can still connect to it to push code or communicate over the serial connection by using a supplied adaptor.)

OPENNESS

The Arduino project is completely open hardware and an open hardware success story.

The only part of the project protected is the Arduino trademark, so they can control the quality of any boards calling themselves an Arduino. In addition to the code being available to download freely, the circuit board schematics and even the EAGLE PCB design files are easily found on the Arduino website.

This culture of sharing has borne fruit in many derivative boards being produced by all manner of people. Some are merely minor variations on the main Arduino Uno, but many others introduce new features or form factors that the core Arduino team have overlooked.

In some cases, such as with the wireless-focused Arduino Fio board, what starts as a third-party board (it was originally the Funnel IO) is later adopted as an official Arduino-approved board.

Arduino Case Study: The Good Night Lamp

While at the IDII, Alexandra Deschamps-Sonsino came up with the idea of an Internet-connected table or bedside lamp. A simple, consumer device, this lamp would be paired with another lamp anywhere in the world, allowing it to switch the other lamp on and off, and vice versa. Because light is integrated into our daily routine, seeing when our loved ones turn, for example, their bedside lamp on or off gives us a calm and ambient view onto their lives.

This concept was ahead of its time in 2005, but the project has now been spun into its own company, the Good Night Lamp. The product consists of a "big lamp" which is paired with one or more "little lamps". The big lamp has its own switch and is designed to be used like a normal lamp. The little lamps, however, don't have switches but instead reflect the state of the big lamp.

Adrian was involved since the early stages as Chief Technology Officer. Adrian and the rest of the team's familiarity with Arduino led to it being an obvious choice as the prototyping platform. In addition, as the lamps are designed to be a consumer product rather than a technical product, and are targeted at a mass market, design, cost, and ease of use are also important. The Arduino platform is simple enough that it is possible to reduce costs and size substantially by choosing which components you need in the production version.

A key challenge in creating a mass-market connected device is finding a convenient way for consumers, some of whom are non-technical, to connect the device to the Internet. Even if the user has WiFi installed, entering authentication details for your home network on a device that has no keyboard or screen presents challenges. As well as looking into options for the best solution for this issue, the Good Night Lamp team are also building a version which connects over the mobile phone networks via GSM or 3G. This option fits in with the team's vision of connecting people via a "physical social network", even if they are not otherwise connected to the Internet.

Arduino Case Study: Botanicalls

Botanicalls (`www.botanicalls.com/`) is a collaboration between technologists and designers that consists of monitoring kits to place in plant pots. The Botanicalls kits then contact the owner if the plant's soil gets too dry. The project write-up humourously refers to this as "an effort to promote successful inter-species understanding" and as a way of translating between a plant's communication protocols (the colour and drooping of leaves) to human protocols, such as telephone, email, or Twitter.

The original project used stock Arduino controllers, although the kits available for sale today use the ATmega 168 microcontroller with a custom board, which remains Arduino-compatible, and the programming is all done using the Arduino IDE. To match the form factor of the leaf-shaped printed circuit board (PCB), the device uses a WizNet Ethernet chip instead of the larger Arduino Ethernet Shield. Future updates might well support WiFi instead.

Arduino Case Study: BakerTweet

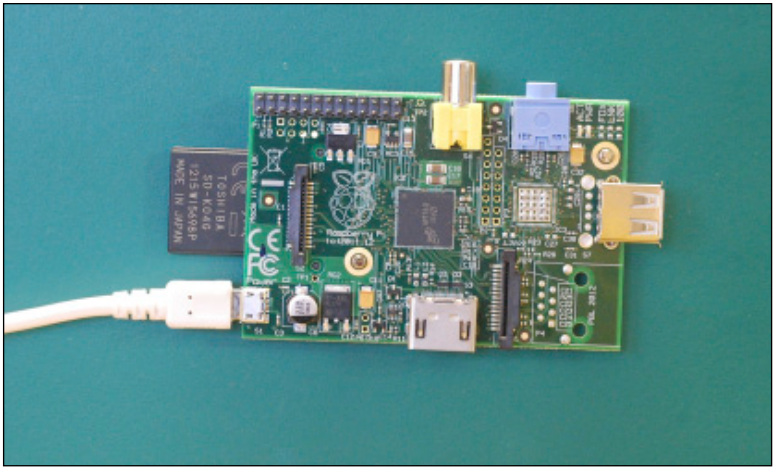
The BakerTweet device (`www.bakertweet.com/`) is effectively a physical client for Twitter designed for use in a bakery. A baker may want to let customers know that a certain product has just come out of the ovens—fresh bread, hot muffins, cupcakes laden with icing—yet the environment he would want to tweet from contains hot ovens, flour dust, and sticky dough and batter, all of which would play havoc with the electronics, keyboard, and screen of a computer, tablet, or phone. Staff of design agency Poke in London wanted to know when their local bakery had just produced a fresh batch of their favourite bread and cake, so they designed a proof of concept to make it possible.

Because BakerTweet communicates using WiFi, bakeries, typically not built to accommodate Ethernet cables, can install it. BakerTweet exposes the functionality of Twitter in a "bakery-proof" box with more robust electronics than a general-purpose computer, and a simplified interface that can be used by fingers covered in flour and dough. It was designed with an Arduino, an Ethernet Shield, and a WiFi adapter. As well as the Arduino simply controlling a third-party service (Twitter), it is also hooked up to a custom service which allows the baker to configure the messages to be sent.

RASPBERRY PI

The Raspberry Pi, unlike the Arduino, wasn't designed for physical computing at all, but rather, for education. The vision of Eben Upton, trustee and cofounder of the Raspberry Pi Foundation, was to build a computer that was small and inexpensive and designed to be programmed and experimented with, like the ones he'd used as a child, rather than to passively consume games on. The Foundation gathered a group of teachers, programmers, and hardware experts to thrash out these ideas from 2006.

While working at Broadcom, Upton worked on the Broadcom BCM2835 system-on-chip, which featured an exceptionally powerful graphics processing unit (GPU), capable of high-definition video and fast graphics rendering. It also featured a low-power, cheap but serviceable 700 MHz ARM CPU, almost tacked on as an afterthought. Upton described the chip as "a GPU with ARM elements grafted on" (`www.gamesindustry.biz/articles/digitalfoundry-inside-raspberry-pi`).



A Raspberry Pi Model B board. The micro USB connector only provides power to the board; the USB connectivity is provided by the USB host connectors (centre-bottom and centre-right).

The project has always taken some inspiration from a previous attempt to improve computer literacy in the UK: the "BBC Micro" built by Acorn in the early 1980s. This computer was invented precisely because the BBC producers tasked with creating TV programmes about programming realised that there wasn't a single cheap yet powerful computer platform that was sufficiently widespread in UK schools to make it a sensible topic for their show. The model names of the Raspberry Pi, "Model A" and "Model B", hark back to the different versions of the BBC Micro. Many of the other trustees of the Raspberry Pi Foundation, officially founded in 2009, cut their teeth on the BBC Micro. Among them was David Braben, who wrote the seminal game of space exploration, Elite, with its cutting-edge 3D wireframe graphics.

Due in large part to its charitable status, even as a small group, the Foundation has been able to deal with large suppliers and push down the costs of the components. The final boards ended up costing around £25 for the more powerful Model B (with built-in Ethernet connection). This is around the same price point as an Arduino, yet the boards are really of entirely different specifications.

The following table compares the specs of the latest, most powerful Arduino model, the Due, with the top-end Raspberry Pi Model B:

	Arduino Due	Raspberry Pi Model B
CPU Speed	84 MHz	700 MHz ARM11
GPU	None	Broadcom Dual-Core VideoCore IV Media Co-Processor

	Arduino Due	Raspberry Pi Model B
RAM	96KB	512MB
Storage	512KB	SD card (4GB +)
OS	Bootloader	Various Linux distributions, other operating systems available
Connections	54 GPIO pins 12 PWM outputs 4 UARTs SPI bus I²C bus USB 16U2 + native host 12 analogue inputs (ADC) 2 analogue outputs (DAC)	8 GPIO pins 1 PWM output 1 UART SPI bus with two chip selects I²C bus 2 USB host sockets Ethernet HDMI out Component video and audio out

So, the Raspberry Pi is effectively a computer that can run a real, modern operating system, communicate with a keyboard and mouse, talk to the Internet, and drive a TV/monitor with high-resolution graphics. The Arduino has a fraction of the raw processing power, memory, and storage required for it to run a modern OS. Importantly, the Pi Model B has built-in Ethernet (as does the Arduino Ethernet, although not the Due) and can also use cheap and convenient USB WiFi dongles, rather than having to use an extension "shield".

Note that although the specifications of the Pi are in general more capable than even the top-of-the-range Arduino Due, we can't judge them as "better" without considering what the devices are for! To see where the Raspberry Pi fits into the Internet of Things ecosystem, we need to look at the process of interacting with it and getting it to do useful physical computing work as an Internet-connected "Thing", just as we did with the Arduino! We look at this next.

However, it is worth mentioning that a whole host of devices is available in the same target market as the Raspberry Pi: the Chumby Hacker Board, the BeagleBoard, and others, which are significantly more expensive. Yes, they may have slightly better specifications, but for the price difference, there may seem to be very few reasons to consider them above the Raspberry Pi. Even so, a project might be swayed by existing hardware, better tool support for a specific chipset, or ease-of-use considerations. In an upcoming section, we look at one such board, the BeagleBone, with regards to these issues.

CASES AND EXTENSION BOARDS

Still, due to the relative excitement in the mainstream UK media, as well as the usual hacker and maker echo chambers, the Raspberry Pi has had some

real focus. Several ecosystems have built up around the device. Because the Pi can be useful as a general-purpose computer or media centre without requiring constant prototyping with electronic components, one of the first demands enthusiasts have had was for convenient and attractive cases for it. Many makers blogged about their own attempts and have contributed designs to Thingiverse, Instructables, and others. There have also been several commercial projects. The Foundation has deliberately not authorised an "official" one, to encourage as vibrant an ecosystem as possible, although staffers have blogged about an early, well-designed case created by Paul Beech, the designer of the Raspberry Pi logo (`http://shop.pimoroni.com/products/pibow`).

Beyond these largely aesthetic projects, extension boards and other accessories are already available for the Raspberry Pi. Obviously, in the early days of the Pi's existence post launch, there are fewer of these than for the Arduino; however, many interesting kits are in development, such as the Gertboard (`www.raspberrypi.org/archives/tag/gertboard`), designed for conveniently playing with the GPIO pins.

Whereas with the Arduino it often feels as though everything has been done already, in the early days of the Raspberry Pi, the situation is more encouraging. A lot of people are doing interesting things with their Pis, but as the platform is so much more high level and capable, the attention may be spread more thinly—from designing cases to porting operating systems to working on media centre plug-ins. Physical computing is just *one* of the aspects that attention may be paid to.

DEVELOPING ON THE RASPBERRY PI

Whereas the Arduino's limitations are in some ways its greatest feature, the number of variables on the Raspberry Pi are much greater, and there is much more of an emphasis on being able to do things in alternative ways. However, "best practices" are certainly developing. Following are some suggestions at time of writing. (It's worth checking on the Raspberry Pi websites, IRC channels, and so on, later to see how they will have evolved.)

> *If you want to seriously explore the Raspberry Pi, you would be well advised to pick up a copy of the* Raspberry Pi User Guide, *by Eben Upton and Gareth Halfacree (Wiley, 2012).*

Operating System

Although many operating systems can run on the Pi, we recommend using a popular Linux distribution, such as

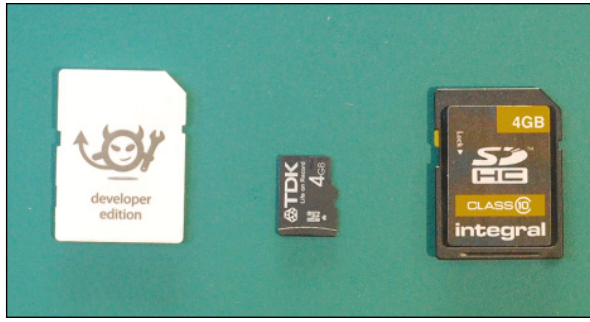
- **Raspbian:** Released by the Raspbian Pi Foundation, Raspbian is a distro based on Debian. This is the default "official" distribution and is certainly a good choice for general work with a Pi.
- **Occidentalis:** This is Adafruit's customised Raspbian. Unlike Raspbian, the distribution assumes that you will use it "headless"—not connected to keyboard and monitor—so you can connect to it remotely by default. (Raspbian requires a brief configuration stage first.)

For Internet of Things work, we recommend something such as the Adafruit distro. You're most probably not going to be running the device with a keyboard and display, so you can avoid the inconvenience of sourcing and setting those up in the first place. The main tweaks that interest us are that

- The sshd (SSH protocol daemon) is enabled by default, so you can connect to the console remotely.
- The device registers itself using zero-configuration networking (zero-conf) with the name `raspberrypi.local`, so you don't need to know or guess which IP address it picks up from the network in order to make a connection.

When we looked at the Arduino, we saw that perhaps the greatest win was the simplicity of the development environment. In the best case, you simply downloaded the IDE and plugged the device into the computer's USB. (Of course, this elides the odd problem with USB drivers and Internet connection when you are doing Internet of Things work.) With the Raspberry Pi, however, you've already had to make decisions about the distro and download it. Now that distro needs to be unpacked on the SD card, which you purchase separately. You should note that some SD cards don't work well with the Pi; apparently, "Class 10" cards work best. The class of the SD card isn't always clear from the packaging, but it is visible on the SD card with the number inside a larger circular "C".

At this point, the Pi may boot up, if you have enough power to it from the USB. Many laptop USB ports aren't powerful enough; so, although the "On" light displays, the device fails to boot. If you're in doubt, a powered USB hub seems to be the best bet.



An Electric Imp (left), next to a micro SD card (centre), and an SD card (right).

After you boot up the Pi, you can communicate with it just as you'd communicate with any computer—that is, either with the keyboard and monitor that you've attached, or with the Adafruit distro, via ssh as mentioned previously. The following command, from a Linux or Mac command line, lets you log in to the Pi just as you would log in to a remote server:

```
$ ssh root@raspberrypi.local
```

From Windows, you can use an SSH client such as PuTTY (`www.chiark.greenend.org.uk/~sgtatham/putty/`). After you connect to the device, you can develop a software application for it as easily as you can for any Linux computer. How easy that turns out to be depends largely on how comfortable you are developing for Linux.

Programming Language

One choice to be made is which programming language and environment you want to use. Here, again, there is some guidance from the Foundation, which suggests Python as a good language for educational programming (and indeed the name "Pi" comes initially from Python).

Let's look at the "Hello World" of physical computing, the ubiquitous "blinking lights" example:

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setmode(GPIO.BOARD) # set the numbering scheme to be the
                         # same as on the board
GPIO.setup(8, GPIO.OUT)  # set the GPIO pin 8 to output mode

led = False
```

```
GPIO.output(8, led)  # initiate the LED to off

while 1:
    GPIO.output(8, led)
    led = not led # toggle the LED status on/off for the next
                  # iteration
    sleep(10)     # sleep for one second
```

As you can see, this example looks similar to the C++ code on an Arduino. The only real differences are the details of the modularization: the GPIO code and even the `sleep()` function have to be specified. However, when you go beyond this level of complexity, using a more expressive "high-level" language like Python will almost certainly make the following tasks easier:

- Handling strings of character data
- Completely avoiding having to handle memory management (and bugs related to it)
- Making calls to Internet services and parsing the data received
- Connecting to databases and more complex processing
- Abstracting common patterns or complex behaviours

Also, being able to take advantage of readily available libraries on PyPi (`https://pypi.python.org/pypi`) may well allow simple reuse of code that other people have written, used, and thoroughly tested.

So, what's the catch? As always, you have to be aware of a few trade-offs, related either to the Linux platform itself or to the use of a high-level programming language. Later, where we mention "Python", the same considerations apply to most higher-level languages, from Python's contemporaries Perl and Ruby, to the compiled VM languages such as Java and C#. We specifically contrast Python with C++, as the low-level language used for Arduino programming.

- **Python, as with most high-level languages, compiles to relatively large (in terms of memory usage) and slow code, compared to C++.** The former is unlikely to be an issue; the Pi has more than enough memory. The speed of execution may or may not be a problem: Python is likely to be "fast enough" for most tasks, and certainly for anything that involves talking to the Internet, the time taken to communicate over the network is the major slowdown. However, if the electronics of the sensors and actuators you are working with require split-second timing, Python *might* be too slow. This is by no means certain; if Bubblino starts blowing bubbles a millisecond later, or the DoorBot unlocks the office a millisecond after you scan your RFID card to authenticate, this delay may be acceptable and not even noticeable.

- **Python handles memory management automatically.** Because handling the precise details of memory allocation is notoriously fiddly, automatic memory management generally results in fewer bugs and performs adequately. However, this automatic work has to be scheduled in and takes some time to complete. Depending on the strategy for garbage collection, this may result in pauses in operation which might affect timing of subsequent events.

  Also, because the programmer isn't exposed to the gory details, there may well be cases in which Python quite reasonably holds onto more memory than you might have preferred had you been managing it by hand. In worse cases, the memory may never be released until the process terminates: this is a so-called *memory leak*. Because an Internet of Things device generally runs unattended for long periods of time, these leaks may build up and eventually end up with the device running out of memory and crashing. (In reality, it's more likely that such memory leaks happen as a result of programming error in manual memory management.)
- **Linux itself arguably has some issues for "real-time" use.** Due to its being a relatively large operating system, with many processes that may run simultaneously, precise timings may vary due to how much CPU priority is given to the Python runtime at any given moment. This hasn't stopped many embedded programmers from moving to Linux, but it may be a consideration for your case.
- **An Arduino runs only the one set of instructions, in a tight loop, until it is turned off or crashes.** The Pi constantly runs a number of processes. If one of these processes misbehaves, or two of them clash over resources (memory, CPU, access to a file or to a network port), they may cause problems that are entirely unrelated to your code. This is unlikely (many well-run Linux computers run without maintenance for years and run businesses as well as large parts of the Internet) but may result in occasional, possibly intermittent, issues which are hard to identify and debug.

We certainly don't want to put undue stress on the preceding issues! They are simply trade-offs that may or may not be important to you, or rather more or less important than the features of the Pi and the access to a high-level programming language.

The most important issue, again, is probably the ease of use of the environment. If you're comfortable with Linux, developing for a Pi is relatively simple. But it doesn't approach the simplicity of the Arduino IDE. For example, the Arduino starts your code the moment you switch it on. To get the same behaviour under Linux, you could use a number of mechanisms, such as an initialisation script in `/etc/init.d`.

First, you would create a wrapper script—for example, `/etc/init.d/StartMyPythonCode`. This script would start your code if it's called with a `start` argument, and stop it if called with `stop`. Then, you need to use the `chmod` command to mark the script as something the system can run: `chmod +x /etc/init.d/StartMyPythonCode`. Finally, you register it to run when the machine is turned on by calling `sudo update-rc.d StartMyPythonCode defaults`.

If you are familiar with Linux, you may be familiar with this mechanism for automatically starting services (or indeed have a preferred alternative). If not, you can find tutorials by Googling for "Raspberry Pi start program on boot" or similar. Either way, although setting it up isn't *hard* per se, it's much more involved than the Arduino way, if you aren't already working in the IT field.

Debugging

While Python's compiler also catches a number of syntax errors and attempts to use undeclared variables, it is also a relatively permissive language (compared to C++) which performs a greater number of calculations at runtime. This means that additional classes of programming errors won't cause failure at compilation but will crash the program when it's running, perhaps days or months later.

Whereas the Arduino had fairly limited debugging capabilities, mostly involving outputting data via the serial port or using side effects like blinking lights, Python code on Linux gives you the advantages of both the language and the OS. You could step through the code using Python's integrated debugger, attach to the process using the Linux `strace` command, view logs, see how much memory is being used, and so on. As long as the device itself hasn't crashed, you may be able to ssh into the Raspberry Pi and do some of this debugging while your program has failed (or is running but doing the wrong thing).

Because the Pi is a general-purpose computer, without the strict memory limitations of the Arduino, you can simply use `try... catch...` logic so that you can trap errors in your Python code and determine what to do with them. For example, you would typically take the opportunity to log details of the error (to help the debugging process) and see if the unexpected problem can be dealt with so that you can continue running the code. In the worst case, you might simply stop the script running and have it restart again afresh!

Python and other high-level languages also have mature testing tools which allow you to assert expected behaviours of your routines and test that they

perform correctly. This kind of automated testing is useful when you're working out whether you've finished writing correct code, and also can be rerun after making other changes, to make sure that a fix in one part of the code hasn't caused a problem in another part that was working before.

SOME NOTES ON THE HARDWARE

The Raspberry Pi has 8 GPIO pins, which are exposed along with power and other interfaces in a 2-by-13 block of male header pins. Unlike those in the Arduino, the pins in the Raspberry Pi aren't individually labelled. This makes sense due to the greater number of components on the Pi and also because the expectation is that fewer people will use the GPIO pins and you are discouraged from soldering directly onto the board. The intention is rather that you will plug a cable (IDC or similar) onto the whole block, which leads to a "breakout board" where you do actual work with the GPIO.

Alternatively, you can connect individual pins using a female jumper lead onto a breadboard. The pins are documented on the schematics. A female-to-male would be easiest to connect from the "male" pin to the "female" breadboard. If you can find only female-to-female jumpers, you can simply place a header pin on the breadboard or make your own female-to-male jumper by connecting a male-to-male with a female-to-male! These jumpers are available from hobbyist suppliers such as Adafruit, Sparkfun, and Oomlout, as well as the larger component vendors such as Farnell.

The block of pins provides both 5V and 3.3V outputs. However, the GPIO pins themselves are only 3.3V tolerant. The Pi doesn't have any over-voltage protection, so you are at risk of breaking the board if you supply a 5V input! The alternatives are to either proceed with caution or to use an external breakout board that has this kind of protection. At the time of writing, we can't recommend any specific such board, although the Gertboard, which is mentioned on the official site, looks promising.

Note that the Raspberry Pi doesn't have any analogue inputs (ADC), which means that options to connect it to electronic sensors are limited, out of the box, to digital inputs (that is, on/off inputs such as buttons). To get readings from light-sensitive photocells, temperature sensors, potentiometers, and so on, you need to connect it to an external ADC via the SPI bus. You can find instructions on how to do this at, for example, `http://learn.adafruit.com/reading-a-analog-in-and-controlling-audio-volume-with-the-raspberry-pi/overview`.

We mentioned some frustrations with powering the Pi earlier: although it is powered by a standard USB cable, the voltage transmitted over USB from a

laptop computer, a powered USB hub, or a USB charger varies greatly. If you're not able to power or to boot your Pi, check the power requirements and try another power source.

OPENNESS

Because one of the goals of the Raspberry Pi is to create something "hackable", it is no surprise that many of the components are indeed highly open: the customised Linux distributions such as "Raspbian" (based on Debian), the ARM VideoCore drivers, and so on. The core Broadcom chip itself is a proprietary piece of hardware, and they have released only a partial datasheet for the BCM2835 chipset. However, many of the Raspberry Pi core team are Broadcom employees and have been active in creating drivers and the like for it, which are themselves open source.

These team members have been able to publish certain materials, such as PDFs of the Raspberry Pi board schematics, and so on. However, the answer to the question "Is it open hardware?" is currently "Not yet" (`www.raspberrypi.org/archives/1090#comment-20585`).

Raspberry Pi Case Study: DoES Liverpool's DoorBot

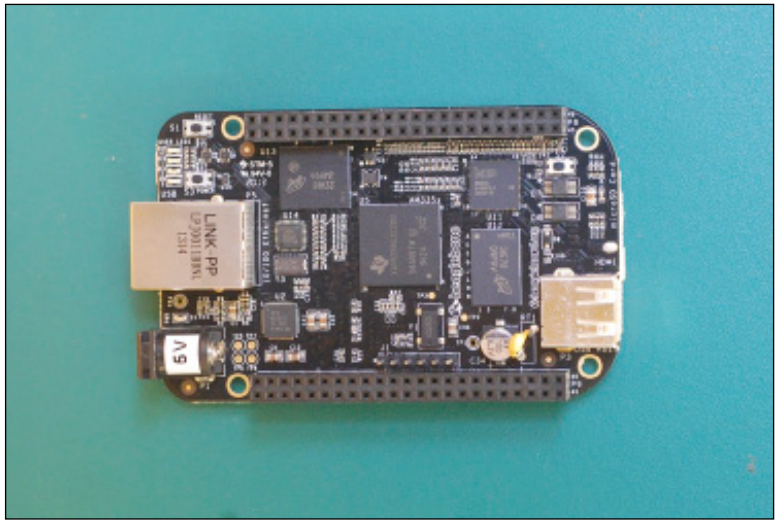
When a group of friends and collaborators, including us, fell upon the idea of setting up a coworking and maker space, we soon discovered that keeping the space open required one of the keyholders to get in early enough and stay late enough to make it attractive to members. Between a big enough group of unpaid organisers, things work reasonably well, but after running for a year and a half, John and Ben extended the DoorBot, which we mentioned in Chapter 4, "Thinking About Prototyping", to managing door entry using a magnetically controlled lock.

Now running on a Raspberry Pi, three DoorBots each control one door (to the main space, or to the office and event spaces). They take input from an RFID pad and check it against a database of known IDs. On a successful entry, the DoorBot plays a piece of "entrance music" specific to that user, triggers a command to the magnetic door bolt to open for 3 seconds, and logs the user's entrance. This solution required fitting new locks with control wires and also drilling holes through the wall panel to feed the RFID reader through to the outside (the affordable reader we used isn't powerful to read through glass). The DoorBot also runs a screen which shows a calendar of events and some webcam feeds. The audio/visual requirements are the main reason that the Raspberry Pi was chosen to implement it. However, having a general-purpose Linux system has also been useful in general: in the case of the entry system code failing, it is possible to connect remotely over ssh to get the door open.

It is worth noting that the Broadcom chip is currently harder to source than either the widely available Atmel chips in the Arduino or the Texas Instruments chip in the BeagleBone. This could make it harder to spin up a prototype into a product.

BEAGLEBONE BLACK

The BeagleBone Black is the latest device to come from the BeagleBoard group. This group largely consists of employees of Texas Instruments, and although the products are not TI boards as such, they use many components with their employer's blessing. The relationship is thus similar to that of the Raspberry Pi Foundation with Broadcom. Similarly, the BeagleBoard team want to create "powerful, open, and embedded devices" with a goal of contributing to the open source community, including facilitating education in electronics. However, there is less of an emphasis on creating a general-purpose computer for education; these boards are very much designed with the expectation that they will be used for physical computing and experimentation in electronics.



The latest board from the BeagleBone family: the BeagleBone Black.

The BeagleBone Black is the smallest and cheapest of the team's boards, with a form factor comparable to that of the Raspberry Pi. Although the specs of the two are mostly comparable, there are some interesting trade-offs.

The original BeagleBone has no video or audio outputs built in, but it does have a far larger number of GPIO pins, extracted into two rows of female

headers. It also has the crucial ADC pins for analogue input which the Raspberry Pi lacks. This shows the development team's focus on building something for working with electronics rather than as a general-purpose computer.

The BeagleBone was released before the Raspberry Pi, and its price reflects that. If you think of it as a more powerful embedded development board than any of the Arduino offerings, then a £63 price tag looks quite reasonable. When you compare it to a £25 Raspberry Pi, however, it makes less sense.

The influence of the Pi can be seen in the latest revision of the BeagleBone platform, the BeagleBone Black. Although still missing the analogue video and audio connectors, it adds a micro-HDMI connector to provide digital outputs for both audio and video. The price is also much closer at £31, and it retains the much better electronics interfacing capabilities of the original BeagleBone.

Let's compare the specs of the Raspberry Pi Model B with the new BeagleBone Black:

	BeagleBone Black	**Raspberry Pi Model B**
CPU Speed	1GHz ARM Cortex-A8	700 MHz ARM11
GPU	SGX530 3D	Broadcom Dual-Core VideoCore IV Media Co-Processor
RAM	512MB	512MB
Storage	2GB embedded MMC, plus uSD card	SD card (4GB +)
OS	Various Linux distributions, Android, other operating systems available	Various Linux distributions, other operating systems available
Connections	65 GPIO pins, of which 8 have PWM 4 UARTs SPI bus I²C bus USB client + host Ethernet Micro-HDMI out 7 analog inputs (ADC) CAN bus	8 GPIO pins, of which 1 has PWM 1 UART SPI bus with two chip selects I²C bus 2 USB host sockets Ethernet HDMI out Component video and audio out

Like the Raspberry Pi, the BeagleBone is a computer that mostly runs Linux but is capable of running a variety of other ported operating systems. Unlike the Pi, it isn't specified by default to communicate with a local user via a display and keyboard.

Importantly, for Internet of Things work, both boards come with Ethernet connectivity (assuming the Raspberry Pi Model B) and can take advantage of cheap USB WiFi dongle options if required.

If the actual advantages of the BeagleBone were limited to a slightly faster CPU and (in our opinion) a more aesthetically pleasing physical design, these might not be compelling reasons to dedicate a section of this book to the platform, never mind actually buy one, given its price.

Let's now look at the features of the BeagleBone that we think distinguish it, including both hardware features and the softer, usability-related ones.

CASES AND EXTENSION BOARDS

Although the BeagleBone hasn't had the same hype and media exposure as the Pi, a fair number of attractive cases are available, either sold commercially like the Adafruit BoneBox, or as freely available instructions and designs. In any case, given that the board is mostly designed to be used for open hardware, it is also likely that the specific project you are working on will require a custom casing to fit the form factor of the project.

Extension boards for the BeagleBone are known as "capes" rather than "shields" (the term for Arduino extension boards). This name comes from Underdog, the star of a 1960s US animation, a beagle who wore a superhero cape. Capes are available to add controllers for LCD displays and motors and various networking and sensor applications.

DEVELOPING ON THE BEAGLEBONE

We've already seen how the Arduino had one official way to do things, whereas the Raspberry Pi presented a blank slate and choices for operating system and programming language. Yes, you can skip the IDE entirely and use `avr-gcc` yourself and write code and build a toolchain to target the Arduino's AVR chip directly, rather than using the Arduino libraries; however, almost all users will at least start with the "official" development tools, and only a rare few will experiment beyond this. In contrast, although recommendations and best practices exist for the Raspberry Pi, you have to make an explicit decision which set of tools to use and even take a step to

install them on the SD card. This flexibility makes some sense because the Pi's target audience is so wide (from education, general-purpose computing, programming, electronics, and home media).

Although the BeagleBone is capable of general-purpose computing, its target market is much more narrowly defined, and perhaps as a consequence, the team have been able to take a middle way: every board comes with the Ångström Linux distribution installed on it. Although several other operating systems have been ported to the platform, having a default installed with zero effort means that it is far simpler to get started and evaluate the board. If you enjoy tinkering, have a preferred embedded operating system, or have reached some insurmountable limitation of Ångström, by all means install another operating system! However, in this chapter we just look at the default.

The BeagleBone runs zeroconf networking by default and usually advertises itself as `beaglebone.local`. You can connect to it in a number of ways. Very usefully, you can connect to the excellent, interactive System Reference Manual and Hardware Documentation, which is generally at `http://beaglebone.local/README.htm`. You can also find this manual at `http://beagleboard.org/static/beaglebone/latest/README.htm`, although it is not usefully interactive in this case.

The manual is dynamically updated to give information about your BeagleBone and has pages where you can look at and change various settings, such as the outputs to GPIO pins. It also has links to the Cloud9 IDE, which is hosted on the board, and which is the simplest recommended way to write code for the device.

Cloud9 is an online programming environment—and a rather interesting idea. A hosted version of it is available at `https://c9.io`, where you can edit and code "in the cloud" (that is, as an Internet-hosted service). However, the primary component of the service is an open source application which can run locally on a given computer (`https://github.com/ajaxorg/cloud9/`). Because the BeagleBone is essentially a general-purpose computer, using the latter, stand-alone option makes more sense. This means that to develop on the BeagleBone, you can get started immediately, without needing to download any software at all, by connecting to the IDE at `http://beaglebone.local:3000`.

Although the online Cloud9 environment also supports Ruby and Python, the free component version supports only Node.js, a framework built on JavaScript. Here's what it says on the tin:

> *Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.*
>
> —`http://nodejs.org/`

We've already addressed the trade-offs of "real-time" platforms compared to Python running on the Raspberry Pi. Many of the same factors apply to the Node.js platform, so let's just summarise those here:

- Although the Chrome JavaScript runtime is aggressively optimised, it is a virtual machine running a very high-level, dynamic language, with greater memory usage.
- Automated memory management has overheads which may affect timing.
- Linux itself is less suited to very precise timings than a true RTOS.

Another important trade-off when looking at real-time programming is the multitasking model. If more than one task needs to be done "at the same time" (or apparently at the same time), such as moving a motor, sensing whether a button has been pressed, and blinking multiple lights in different patterns, the runtime environment needs a strategy to choose when to switch between the "simultaneous" tasks. Node.js uses a cooperative model, whereby each action is run as appropriate by a coordinating central process. This process chooses the next action to run based on its schedule. But because only one action actually runs at a time, the coordinator has to wait until the previous action has completed before starting the next one. This means that there is no way to guarantee that a given task will start with millisecond precision.

The fact that the Node.js website mentions "real-time" code suggests that its developers believe their technology is perfectly capable of doing multitasking event scheduling. Although the specific definition of "real-time" may not match that used by some embedded programmers for applications requiring split-millisecond precision, it is "good enough" for most tasks. Besides which, the advantages of the high-level language and convenient development environment often are the overriding concern.

As mentioned previously, the Arduino has the benefit of a single-standard development toolchain, whereas the Raspberry Pi has the double-edged sword of an embarrassment of riches. The BeagleBone combines the best of these two extremes by having all the flexibility of the Pi but with a single standard toolchain installed.

For the rest of this section, we look mostly at the standard toolchain; rest assured, though, that if it doesn't quite live up to your requirements, you could substitute plenty of other options.

Operating System

The BeagleBone comes pre-installed with the Ångström Linux distribution. This operating system was developed specifically for running on embedded devices and can run with as little as 4MB of flash storage.

Much like Occidentalis for the Raspberry Pi, Ångström is configured to run "headless", without keyboard and monitor. There are convenient ways to access a command-line prompt, either over a serial connection with USB or using zeroconf and connecting to `beaglebone.local`.

One advantage of having a single *pre-installed* operating system is that you can start playing with the board the moment you open the box. Of course, the onboard instructions suggest updating the operating system as one of the first steps, if it's not already at the latest version; this is certainly good practice after you have briefly evaluated the board and are moving onto prototyping a project with it.

Programming Language

As with the Raspberry Pi, you can connect to BeagleBone via the terminal and develop a software application as you would for any other Linux computer. In the preceding section, we noted that this is a "simple" task if you already program for Linux, but it might seem a little overwhelming otherwise. In this section, we look at the Cloud9 IDE, which we mentioned earlier, in a little more detail.

As before, we start with the old, reliable "blinking lights" example. The `README.htm` presents code to do exactly this:

```
require('bonescript');

setup = function () {
    pinMode(bone.USR3, OUTPUT);    // Enable LED USR3 control
}

loop = function () {
    digitalWrite(bone.USR3, HIGH); // Turn LED USR3 on
    delay(100);                    // Delay 100ms
    digitalWrite(bone.USR3, LOW);  // Turn LED USR3 off
    delay(100);                    // Delay 100ms
}
```

As always, the code is practically the same in concept: the details of the syntax vary, but you still have the usual two functions, `setup` and `loop`; have to set the mode of the pin to output; and alternate sending `HIGH` and `LOW` values to it with a pause in between.

Note that USR3 is actually connected to one of the status LEDs built into the board. If you want to connect a standard kit LED using the GPIO pins, you would simply substitute `USR3` with one of the standard pins, such as `P8_3`.

As with Python, JavaScript is a high-level language, so after you move beyond the physical input and output with electronic components, tasks dealing with text manipulation, talking to Internet services, and so on are far easier to handle than the relatively low-level C++ used in the Arduino.

And again, Node.js is a rich environment with a host of libraries available to integrate into the app. Currently, the convenient `npm` (Node Packaged Modules) utility isn't bundled with the IDE, but this is an item for a future version. In the meantime, online help and forums should get you over any possible stumbling blocks.

The code doesn't automatically start after a reboot, unlike the Arduino. However, once you have finished developing your application, copying it into the `autorun` subfolder will result in the system's picking it up on reboot and starting it automatically from then on.

Debugging

Just like Python, JavaScript is a permissive language compared to C++. Again, because the language is dynamic, the compiler does not catch several types of programming errors. Luckily, the IDE reports runtime errors quite usefully, so while you are running from that, you can get reasonable diagnostics with minimal effort.

When you move to running the Node.js code automatically as a service, you will probably want to enable logging to catch those errors that don't become evident until days or months later.

As with the Raspberry Pi, you have all the tools available on the Linux operating system to help debug. This is, of course, all the more necessary because Linux + Node.js is a more complicated platform than Arduino, but the flexibility and power are certainly appreciated!

Again, JavaScript has error handling, although it is extended and arguably complicated by Node.js's callback mechanisms. This feature can help handle and recover from unusual situations gracefully.

Node.js also has automated testing tools such as `node-unit`. Just because you are writing in a high-level language doesn't mean that all your code will be simpler. The fact that it is easier to write any given line often means that you write much more code and of greater complexity. Because Node.js is built on collaborative multitasking through callbacks, the execution path may be especially complicated: testing can allow you to make sure that all the expected paths through this code happen, in the right order.

SOME NOTES ON THE HARDWARE

For physical computing, the number of GPIO pins available is greatly superior to the Pi's, and a number of them have ADCs (allowing them to process analogue inputs, a capability which the Pi lacked), and others offer PWM capability (allowing the board to mimic analogue output). Although the pins are neatly laid along both long edges of the board, in two columns, the BeagleBone doesn't have enough space to label each pin, as the Arduino does. Instead, each set of headers has labels, and the documentation has a key for what each of the physical pins does. As with the Pi, each pin may have a number of different names, depending on which subsystem is accessing it, and this may feel unnecessarily complex if you are used to the simple, restricted naming (and labelling) conventions of the Arduino.

OPENNESS

One of the initial explicit goals of the BeagleBoard was to create an open platform for experimenting with embedded hardware. As such, the project has, to date, released more of its schematics as open source than has the Raspberry Pi Foundation. Specifically, "All hardware from BeagleBoard.org is open source, so you can download the design materials including schematics, bill-of-materials and PCB layout" (`http://beagleboard.org/`). The Ångström Linux distribution that is installed by default on the BeagleBone is also fully open source. Most of the nontechnical content of the site is also released under a Creative Commons licence. Although the Pi team also appear to be pro–open source, the explicit release of material under open source licence seems to be an issue that the BeagleBoard team are taking more seriously. This might be important to your project for philosophical reasons, but open schematics and code may potentially be critical at the stage of moving your project from prototype to product.

The BeagleBone web pages also link to proprietary operating systems and extension boards. As you may expect, we consider this to be a good thing, as the team are encouraging a good ecosystem around the product, both free and commercial.

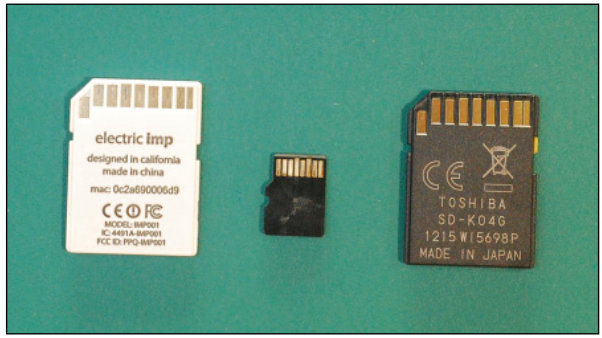
BeagleBone Case Study: Ninja Blocks

Although the rise of the cheap microcontrollers we have been looking at brings Internet of Things applications such as home automation into the price range of an interested hobbyist, and the progressively simpler development environments put it within the reach of those with little or no programming experience, it is still true that developing useful prototypes with an Arduino, Raspberry Pi, BeagleBone, and the like does require getting your hands dirty with the awkward details of hardware and software. The Ninja Blocks (`http://ninjablocks.com/`) are one solution to this problem. Just as "If This Then That" offers simple rules-based programming for web applications—"If I am tagged in a picture on Facebook, Then upload that picture to Dropbox"—the Ninja Rules app extends this idea to Internet of Things devices: "If my washing machine has finished, Then send me an email".

The brains of a Ninja Block is a BeagleBone running Linux and packing an Arduino, too (presumably to take advantage of its connectivity, although in general the BeagleBone has equivalent connectivity options). It can be connected to other blocks to provide input and output. Input is provided by sensor blocks, such as motion and door contact sensors for security applications, and buttons and temperature sensors for general home monitoring. Outputs are provided by actuators, currently remote power controls. Although the Rules app provides the simplified use cases mentioned here, you also can program the blocks using a REST API or by using high-level code (such as Node.js) on the BeagleBone or, indeed, low-level C++ on the Arduino.

ELECTRIC IMP

Although we're featuring the Electric Imp here, it's a less mature and, in some ways, more problematic offering than the other boards we've discussed, and we can't tell, at the time of writing, if the platform will develop into a viable choice. But it is worth discussing in some detail, as a possible paradigm shift in the way that developers approach consumer electronics and physical computing.



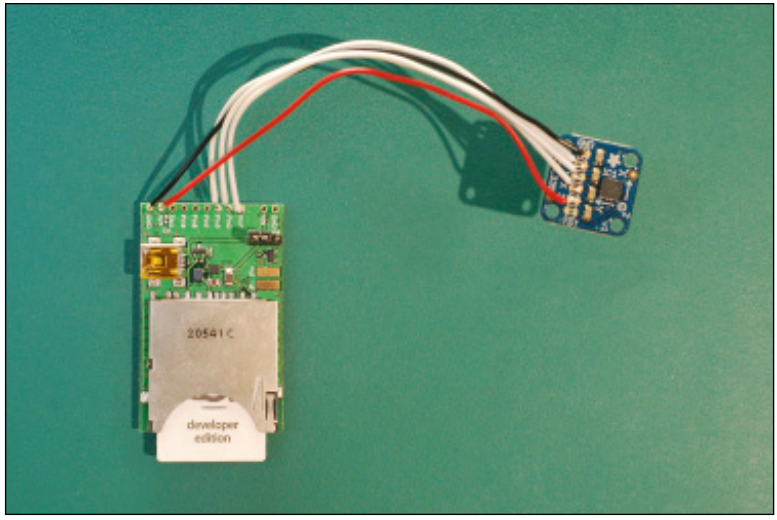
The rear of (from left to right): an Electric Imp, a micro SD card, an SD card.

Hugo Fiennes, formerly engineering manager on Apple's iPhone team, was attempting to connect LED lights to Google's share price. He evaluated various home automation options, like ZigBee, but realised that they were mostly single-vendor solutions, often using their own radio standards rather than based on open platforms (`http://www.edn.com/electronics-news/4373185/Former-Apple-Google-Facebook-engineers-launch-IoT-startup-item-2`). The Electric Imp uses a number of existing standards, such as WiFi, and the form factor of SD cards but ends up being very much less of an open platform than all the other devices that we've looked at in this chapter. Fiennes collaborated on the project with Kevin Fox, a former Gmail designer, and firmware engineer Peter Hartley. As you'll see, the startup feels as though it has much of the DNA (for good and bad) of the beautiful, technically polished walled gardens that are the iPhone and Gmail.

All the smarts of the Electric Imp, and also its WiFi connectivity, are located in an SD card–shaped microcontroller. It's important to note that the Imp isn't actually an SD card; it's just shaped like one. Using the same form factor means that producing the Imps is cheaper because the team can reuse existing cases and tooling, as well as existing component connectors for the *impee* (the name Electric Imp use for the rest of the circuit that the Imp plugs into). This last factor is important, as you see in the upcoming "Openness" section.

Although an SD card feels very robust, it is, on the outside, effectively a small, flat piece of plastic. It offers only one affordance to connect it to anything: namely, plug it into a device. Just as you would insert an SD card into a music player, computer, or printer, you insert an Imp into an impee. This host board provides power, GPIO connections to sensors and actuators, and an ID chip so that the Imp knows which device it's plugged into.

The Imp costs around £20, while an impee costs less than half that. Here, having used the standard SD form factor turns out to be a great choice for the prototyper. For prototyping a number of projects, you need only a single Imp, which can be reused across all the projects. You will see shortly that reconfiguring the Imp to run on a different impee is automatic, which is a very nice feature.



An Electric Imp inserted into an April impee development board and wired up to an accelerometer (the smaller circuit board).

DEVELOPING ON THE ELECTRIC IMP

We saw that the Electric Imp gets its power from the impee. Boards like the standard April impee have a connection that you can solder a battery to, but also have a USB connection, which is convenient while you're developing. But unlike the other boards we've looked at so far, the USB is *only* for power and not for communicating with the device. So you may be wondering how you would go about programming it.

In fact, the Imp is programmed and controlled over the Internet, so all it has to do is connect to your WiFi network with its built-in wireless connectivity. But, of course, you still have to tell it which wireless access point to connect to and most likely what password to use. This is obviously a tricky issue, until you already have a means of communicating with the board. The Electric Imp team have solved this bootstrapping problem using a very clever idea called *BlinkUp*. The Imp has a built-in light sensor (still contained in the tiny SD card format) which is activated whenever the Imp doesn't know how to access a WiFi connection. The Electric Imp smartphone apps currently available for iPhone and recent Android encode the WiFi connection data by simply changing the colour of the screen with an alternating pattern of white and black. You place the phone screen over the Imp to communicate with it, so you are quite literally "flashing" your

microcontroller. If you don't have the appropriate kind of phone, you are slightly stuck (currently, you don't even have a way to encode the data using, say, your computer screen), but we assume that you can borrow a phone if required. Of course, the process of flashing the WiFi data across may fail: the Imp responds with a different pattern of blinking LEDs to help you diagnose this problem.

Writing Code

From the Electric Imp website, a sign-in link enables you to register for the site and then log in to your account (`https://plan.electricimp.com/`). This contains a list of all the device "nodes", which is empty at start. You can edit code in an IDE here.

The language used is *Squirrel*, which is a dynamic language with a C-like syntax. At present, Squirrel is poorly documented and poorly optimised. It is comparable to high-level embeddable scripting languages such as Lua, for which numerous published books are available, such as *Programming in Lua* (`www.lua.org/pil/`), currently in its third edition. Although Squirrel's relative obscurity may change, what will probably remain constant is that the name is overloaded. Unless your search engine, like Google, learns your habits and starts to concentrate on the appropriate articles, searches such as "squirrel performance", and "squirrel numbers" will give less than useful results. Including the term "language" or limiting searches to the `www.squirrel-lang.org/` domain may be useful.

After you write the code, pushing it to the device is one of the great strengths of the Electric Imp platform. When you signed in to the BlinkUp app, as well as giving your WiFi details, you also logged in to your electricimp.com account. This means that after you've flashed the Imp, it knows which account it is tied to. When you plug an Imp into an impee, that host device appears as a node in the planner application, which you access on the electricimp.com website. You can then associate it with one of the pieces of code that you've written. Now, whenever you plug the Imp into the impee, it connects to the web service and downloads the *latest* code onto the device. This means that after you have made a code change, you simply need to eject and reinsert the Imp to refresh the code.

Only at this point does the planner show you any error messages, which shows how dynamic the Squirrel language is; there are no compilation checks.

Although the deployment is very polished, the experience of using the editor and planner is still rather counterintuitive. Again, this issue will most likely improve as the platform matures. Let's see what the blinking light example looks like:

```
// Blink-O-Matic example code

local ledState = 0;

function blink()
{
    // Change state
    ledState = ledState?0:1;
    hardware.pin9.write(ledState);

    // Schedule the next state change in 100ms
    imp.wakeup(0.1, blink);
}

// Configure pin 9 as an open drain output with
// internal pull up
hardware.pin9.configure(DIGITAL_OUT_OD_PULLUP);

// Register with the server
imp.configure("Blink-O-Matic", [], []);

// Start blinking
blink();
```

The first point to note is that you don't have the usual obvious division of labour into `setup` and `loop`. All the setup functions are done within the body of the code, while the `blink` function is explicitly called and tells the Imp to call itself again in 100 milliseconds. Apart from this, most of the usual configuration of a hardware pin and writing alternately high and low values to it are familiar.

The `imp.configure()` call tells the planner details of the code. Here, the only parameter filled in is the descriptive name of the node, but as you can see, you have to supply two additional list parameters (in this case empty). We come back to this call shortly.

Debugging

As you saw, any error messages after deployment or during runtime are shown in the editor window. So are messages output from the Imp using

the `server.show()` call. This feature is really handy, compared to the Arduino, for example, because it's easier to use than the serial console, and you also get useful automatic output from errors that occur while your code is running.

The output of `server.show()` is also sent to the planner screen, where it is shown on the "node" box for the impee. When you have only one impee running, this feature isn't especially impressive, but when you have a number of devices running, this box does provide a nice overview of all your devices' status. As you will now see, the planner screen is crucial to really getting the most out of the Imp.

Planner

Perhaps the most interesting feature of the platform is the way that the nodes can be *connected* to each other on the planner screen. Any node could have outputs, where some data is *output*, not just to be shown on the planner screen but to another node. This second node would then have a corresponding *input* which handles the data in some way. So, for example, one Imp might output the value of a switch, whereas another turns a light on or off in response to that switch. This feature works whether the second Imp is in another room (that is, sharing a WiFi network with the first) or on a different network, possibly in another country entirely. You could implement such a system with some code like this:

```
// NODE 1
local my_output = OutputPort("SwitchOut");
imp.configure("RemoteLightSwitch", [], [my_output]);
// later... switch the light on
my_output.set(1);

// NODE 2
class SwitchIn extends InputPort
{
    type = "integer"
    name = "switch_value"

    function set(value) {
        hardware.pin1.write(value)
    }
}
imp.configure("Light", [Servo()], []);
```

The Electric Imp platform has completely hidden all the details of how the two devices communicate with each other! All you have to focus on are the details of the message that gets passed between them and not the details of the HTTP or other protocol calls.

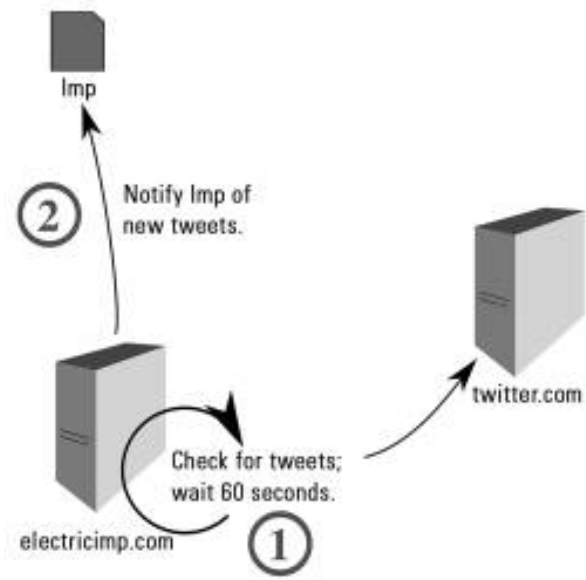
You may have spotted that the `imp.configure` calls in the preceding example have some values provided to the two list parameters that were empty when you last encountered it. This is how the network communication is wired together in the code. The first list shows any inputs (where information is sent from the Electric Imp server to this Imp), and the second list gives the outputs (where this Imp will send data out to the Electric Imp server). Once you have defined some inputs and outputs, you can wire them together in the planner to feed the output from one Imp into the input of another.

So, because communication between Imps doesn't require low-level details such as HTTP communication, the Imp doesn't have a way to make calls to the network. If this were the end of the story, it would obviously make it a terrible choice for any implementation of an Internet of Things product! The planner allows you to add nodes which make HTTP calls. They are run not on the Imp, but in the cloud, on the `plan.electricimp.com` servers. This makes a great deal of sense because the Imps themselves may be running in a network with restrictions on network traffic. Instead, the server application does the processing of the network call and passes JSON data (for example) to the next node.

So what would, in the world of Arduino, Raspberry Pi, or BeagleBone, be code on a single microcontroller, on the Electric Imp platform you would instead decompose the problem into a pipeline of several nodes. Let's look at a few examples of how to implement proof-of-concepts for some of the Internet of Things devices that Adrian and others in our workshop have worked on:

Bubblino:

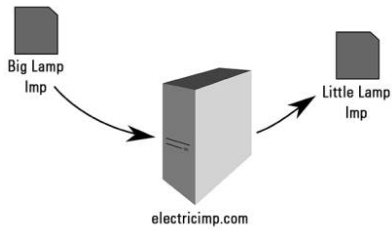
1. The node which communicates with the Twitter servers would live on the Electric Imp server and periodically check for new tweets.
2. Whenever it found one, it would then communicate with the Imp, which triggers the bubble machine to blow bubbles.



Network topology of an Electric Imp implementation of Bubblino.

Good Night Lamp:

This is effectively the example we gave in the preceding text. When the Big Lamp is switched on, its Imp would notify the Little Lamp's Imp, via the Electric Imp server.



A single Big Lamp and Little Lamp for Good Night Lamp would be the simplest network example of Electric Imp.

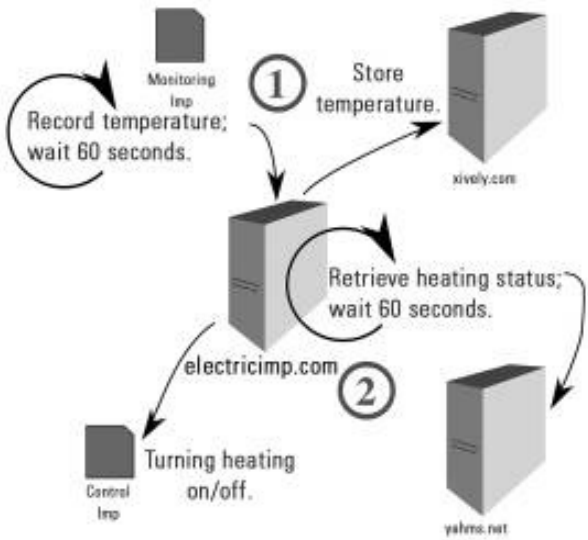
Yet Another Home Management System:

This is the system we use to control the heating in our office. For more details on how it works, see the case study on DoES Liverpool in Chapter 4.

1. The office temperature monitor would be implemented as one Imp, the Monitoring Imp in the following diagram. That would regularly report the temperature to a node on the Electric Imp server, which would record the value to a feed on Xively.
2. Independently of the monitoring system, a control node on the Electric Imp server would poll the server at `yahms.net` to see if the heating should be on or off and notify another Imp, labelled the Control Imp in the diagram. That would then control the heating system and turn it on or off, accordingly.

These examples handwave over details of configuration and control. The planner allows parameters to be tweaked but is too flexible to make it into a commercial product. Of course, the Electric Imp's commercial offering to manufacturers offers more options for this than just this development planner.

Currently, only a handful of server nodes or "virtual impees" are available, but you can find documentation for writing your own server nodes (still in the Squirrel language), and you are encouraged to contact the Electric Imp team if you need access to this capability.



A more complex Electric Imp scenario, to monitor the temperature in an office and allow remote or automatic control of the heating.

Comments on Networking

This transparent networking, where you can join up server nodes with one or more Imps, possibly on multiple networks, is very appealing. But it does mean that you have a central point of communication, and therefore failure, on the Electric Imp team's servers. It also means that two Imps working together in the same room don't network together but via a round-trip to the Internet. This configuration makes the Imp unsuited to tasks in which faster mesh communication would be required and less appealing when you want to expand the communication beyond the reach of your WiFi network—for example, with a line-of-sight infrared or a longer-range radio communication.

Openness

Although you've seen how commercial ecosystems have grown up around the other microcontrollers described in this chapter, the Electric Imp is created not for hobbyists or for education but as a commercial system to be incorporated into products. As such, it isn't optimised for tinkering but rather for getting things done. And unsurprisingly, although the project does use open source elements, the two factors that make up its competitive advantage are, for the moment, resolutely closed source—namely, the innovative form factor and the online planner.

Form Factor

As a small microcontroller enclosed in the plastic casing of an SD card, the Imp doesn't lend itself to tinkering. It has no user-serviceable parts and provides no documentation on how to assemble your own.

Similarly, whatever operating system runs on the Imp is locked down. It runs BlinkUp, the diagnostic LEDs, the communication with the impee to find its ID, and the syncing of its code with the electricimp.com servers. Then it runs the code and communicates, as appropriate, with the server. There is no user-exposed way to change how this works.

Planner

The source code of the planner and the back-end infrastructure that connects Imps and virtual nodes are closed. This situation makes some sense because this is a commercial service; however, it means that when developing for the Imp, you are tied to a vendor to provide the service. You have to trust a third party to keep your code safe, backed up, and available. This isn't a great leap; it may make perfect sense to outsource parts of your business as an IT product, from hosting (Heroku), source control (GitHub), email sending (Postmark), and error logging (Sentry).

However, being reliant on a *single* vendor can be a real issue. The providers mentioned in the preceding paragraph may or may not be the best, long term, but each has viable alternatives. In the case of Electric Imp, which is to some extent blazing a trail, moving your code and devices to another provider if required may not be so simple. You need to be asking questions like these:

- If they close, will it be possible to reconfigure the Imps to point at another provider who can host the code?
- If the level of service is inadequate, what will you do?
- If you get big and want to configure the planning infrastructure in a direction that the Electric Imp team don't want to facilitate for their general customers, how will you proceed? Will they allow you to sponsor the development? Or to take on management of it yourself?

Programming Language

The Squirrel programming language is, in fact, developed under the MIT open source licence. Although it is a relatively new project, compared to other contenders such as Lua, it is possible for you to read the source of Squirrel, fix bugs, add features, and so on. However, as the Electric Imp team are wholly responsible for the version of Squirrel that runs on both the Imps and the virtual nodes, you have no guarantee that such changes will be applied in a timely fashion or indeed ever.

Impees

The reference designs for impees are placed in the public domain (`http://devwiki.electricimp.com/doku.php?id=boards:start`). Although, as a developer, you will probably want to start with the minimal "April" board, which exposes six pins and supplies power, you can easily create your own impee that fits the form factor of your project using the schematics, bill of materials, and Gerber files that the team provide. (And if none of those terms make any sense, fear not; you will find out more in Chapter 10 .)

Here, the decision to use the SD card form factor is also an excellent one. To interface with the Imp, you can buy readily available SD card slots, connect to a cheap Atmel cryptographic chip (to provide a unique ID for the impee) and a few other reasonably priced components, and you have a valid host for an Imp.

The ease of adding the Imp hardware makes real sense for one of the main target audiences of the Electric Imp: existing manufacturers of consumer electronics who want to find a convenient way to incorporate the Internet of Things technologies into it. Finding space for an SD card slot and an ID chip will be much simpler than trying to incorporate a larger board like the Arduino.

Electric Imp Case Study: Lockitron

Although the Raspberry Pi DoorBot solves a very specific problem for a fairly technical group of people, everyone uses doors and keys. Lockitron (`https://lockitron.com/`), from Apigy, is a consumer solution to the problem of granting visitors access to your home without getting more keys cut. As such, it doesn't require changing locks and drilling holes through walls. Rather, a device fits over your current deadbolt and can simply turn the existing lock. You then open the lock using an app on your mobile phone, rather than using RFID (although the Lockitron does also have optional support for NFC radio). This means that the device has to be constantly connected to the Internet: because many people do not have Ethernet cabling running to their front door, this makes wireless connectivity a requirement.

That's where the Electric Imp comes in: "Electric Imp's BlinkUp takes a notoriously difficult WiFi enrollment process and turns it into something simple and delightful. Electric Imp was incredibly straightforward to integrate, and we didn't need to worry about buying expensive toolkits to write our firmware" (`http://blog.electricimp.com/post/45920501558/lockitron`). Although the Imp is one of the simplest microcontrollers we've looked at, because of the specific use case of the Lockitron, it needs to interact only with the lock, sensing whether it has been opened and making it open or close. All the rest of the intelligence happens on the Internet, and is a perfect use case for the Electric Imp's model of division of labour between local impee and cloud nodes.

OTHER NOTABLE PLATFORMS

As we saw in the preceding chapter, any number of prototyping platforms are available, and in the future there will be many more. This chapter has presented our informed guesses on the likely contenders for choice of platform for an Internet of Things project for the next few years. Even if it turns out that we bet on the wrong platform, we hope that the analysis of specific pros and cons of these varied choices will remain useful.

However, we didn't go for several platforms which nevertheless are worth an honourable mention. Let us now briefly survey some of the more interesting options. We've traced a narrative from the low-level Arduino, via the highly

capable but bafflingly flexible Raspberry Pi, to the better packaged and more "opinionated" BeagleBone, and finally the attractive but locked-down Electric Imp. We continue this progression by looking at some devices in which the hardware is entirely locked down: mobile phones and tablets, and plug computing.

There are still ways to get around this closed nature, however, as the explanation of the Android Development Kit (ADK) will demonstrate.

MOBILE PHONES AND TABLETS

Modern phones and tablets, whether they're running iOS (iPhone/iPad), Android, Blackberry, or even Windows, come bristling with sensors, from gyroscopes to thermometers, one or more cameras capable of capturing still images and video, microphones, GPS, WiFi, Bluetooth, USB, NFC (the same technology that runs your contactless bank card or travel pass), buttons, and a multitouch sensitive screen. Phones, of course, have always-on Internet connectivity, via WiFi or the phone network, and so do many tablets. These devices also have several appealing outputs: high-fidelity audio output, HD-quality video, and one or more vibrating elements. They also have processing power similar to the Raspberry Pi's (currently, they are probably faster for general processing but slower for graphics, due to the Pi's crazily powerful GPU.)

On the face of it, using a mobile device sounds like the perfect way to create an Internet of Things appliance. However, the temptation is always to fall back on the device's capabilities as a phone, at which point you are effectively creating an "app" for the device rather than a new "Thing". And although the inputs for a phone can be used remarkably creatively (using the camera to measure your pulse by analysing the changes in redness of your skin), there isn't an easy way to wire a phone up to additional circuits or inputs.

Finally, these devices are still rather large and expensive to be placed into arbitrary consumer devices.

Android Development Kit

Although the cost and size may not matter to you for a prototype, the inability to wire up additional sensors and actuators could stymie any serious use of a mobile device in the Internet of Things. The ADK deals with this problem by pairing an Android device with an ADK-compatible microcontroller (currently the ARM-based Arduino Due). We saw that mobile devices don't have great options for communicating to arbitrary

electronics, but they *do* have a USB connection that is typically used to sync data to a computer.

USB connections traditionally distinguish between the Host (computer) and Device (phone and so on) roles. If you want to connect the Android device to the Arduino, the latter takes on the Host role. In reality, the Android device provides the Internet connection, input/output capabilities, and processing power. The Arduino itself simply offers dumb access to its GPIO pins.

With the cost and size implied by an Android device *and* Arduino Due, not to mention the current state of documentation, the ADK doesn't seem like a viable platform as yet. But the division of labour between an Internet-connected, multimedia-capable device and a physical computing platform is an appealing direction for the future.

PLUG COMPUTING: ALWAYS-ON INTERNET OF THINGS

No matter how small you make a computer, you still have to power it. Even the smallest board may have a power pack that is relatively chunky. The idea of a plug computer is to include the circuitry within the same enclosure as the plug and adaptor. This turns out to be a highly convenient form factor, for a number of reasons:

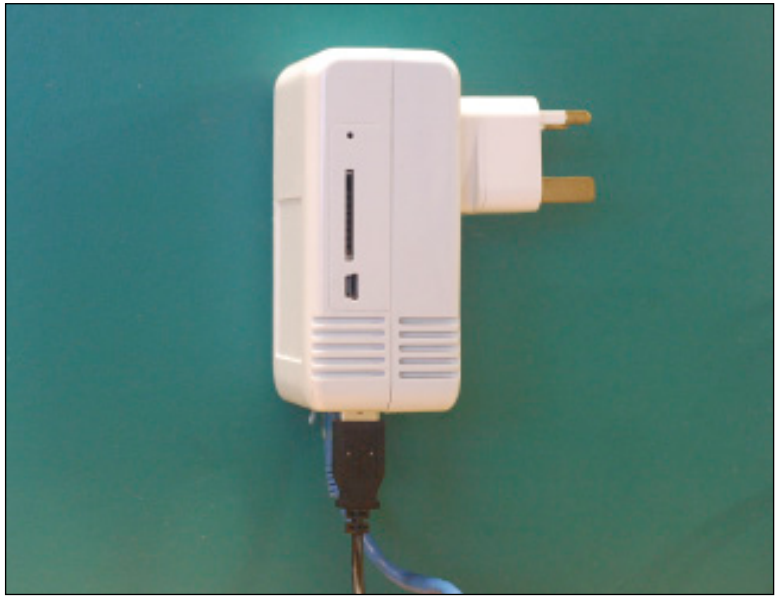
- You don't need to place the computer somewhere.
- It does not have a cable to get dislodged or pulled.
- Being plugged in, tight against a plug socket, the device is unobtrusive.

This last consideration has made the idea of plug computing appealing to, among other groups, security experts and attackers. The Pwn Plug (`http://pwnieexpress.com`) is an "enterprise-class penetration testing drop box" that can be plugged into a socket and test for classic security vulnerabilities. Although this product is advertised for "white-hat hackers" (that is, security consultants helping companies secure their data), similar products could be used for evil. Most employees wouldn't look twice at a new small white box plugged into the wall, if they noticed it at all.

This kind of device, of course, has perfectly legitimate uses. For example, the Freedom Box home office server (`http://p2pfoundation.net/Freedom_Box`) is a cheap and open source way of providing file sharing and email for small companies or your home.

One of the most popular plug computing platforms is the SheevaPlug (`www.plugcomputer.org/`), which typically runs a Debian distribution tailored for its ARM chip. It doesn't have many input/output capabilities, usually only an Ethernet socket and USB.

With the low power requirements of the boards that we've looked at, which can be powered by USB and a tiny adaptor, it should be simple enough to create a plug computer-style enclosure for the Raspberry Pi or BeagleBone which would, in addition, have superior access to GPIO pins. The SheevaPlug and others are attractive in terms of being *already available* consumer electronics rather than necessarily being the best solution for an Internet of Things product.



A SheevaPlug, looking more like a power adapter than a tiny computer.

SUMMARY

By now your ideas about the electronics and embedded software of your project should be starting to coalesce. If you're new to embedded development, don't worry too much about making the "wrong" choice. As you will see in later chapters, the road to a finished Internet of Things product involves some level of reworking of both the electronics and software, so focus on choosing the platform which is best for you *right now* rather than deliberating over a *perfect* choice.

The Electric Imp has great potential to find a niche that we weren't even aware existed: a hosted, walled garden, which is convenient to develop for, and where a commercial entity is responsible for making the right decisions on behalf of its users, who, in return for this capability, happily sacrifice control over some of the finer details. At the time of writing, the platform doesn't feel quite ready yet, but it is one of the more exciting developments, and it, or something like it, may well become the face of a swathe of Internet of Things development in the future.

The Arduino is the established and *de facto* choice, and for good reason. It is unrivalled in the wealth of support and documentation, and its open nature makes it easy to extend and incorporate into a finished product. The only downside to the Arduino is in its more limited capabilities. This simplicity is a boon in most physical computing scenarios, but with many Internet of Things applications requiring good security, the base Arduino Uno platform is looking a little stretched.

The Linux-based systems of the Raspberry Pi and BeagleBone will give you all the processing power and connectivity that you will need, but at the cost of additional complexity and, should you want to take the system into mass production, an additional cost per unit. There isn't much to choose between the Pi and tbe BeagleBone Black when it comes to cost for a prototype, so your choice is more likely to be down to other factors. The Raspberry Pi has a much higher profile, and better community around it, but its greater capability to interface to electronics and easier route to manufacture mean that, in our opinion, the BeagleBone Black has the edge.

With the electronics side covered, the next step is to look at the physical form into which the electronics will sit. The next chapter takes you through the design process, from initial sketches to the tools you can use to bring the design into the world, with a particular focus on the newer digital fabrication options, such as laser cutting and 3D printing.